Diplomarbeit

Cell Computing Model

Autor	Benjamin Hadorn
E-Mail	b_hadorn@bluewin.ch
Ablage/Website	http://www.xatlantis.ch
Datum	20.12.05
Version	1.0.0

Inhaltsverzeichnis

1 Vorwort....7

2 Einleitung....9

3 Was ist Cell Computing Model?....10
3.1 Die Welt der chaotischen Systeme....10
3.1.1 Aufbau der Lebewesen....12
3.1.1.1 Stammzellen....12
3.1.1.2 Spezialisierte Zellen....12
3.1.2 Generierung und Regenerierung....12
3.1.3 Das Programm der Zelle (DNA)....13
3.1.3.1 Wechseln des DNA Abschnitts....13
3.1.4 Das Immunsystem....13
3.2 Ideenkatalog....14
3.2.1 Regenerierung....14
3.2.2 Parallele Verarbeitung der Daten....15
3.2.3 Lernfähigkeit....15
3.3 Zieldefinition....15
3.3.1 Framework....16
3.3.2 Hardware....17
3.4 Einsatzgebiet....17
3.4.1 Vorteile....17
3.4.2 Nachteile....18
3.5 Zukunftsvisionen....19

4 Grundkonzept des CCM....20
4.1 Das EVA-Prinzip....20

5 Das Zellenleben....23
5.1 Der Lebenszyklus....23
5.2 Erzeugen und Abbauen von Zellen....24
5.2.1 Erstellen von Zellen / die Zellenfabriken....24
5.2.2 Klonen von Zellen....25
5.2.2.1 Vermehrung in dem Zellenhaufen....26
5.2.3 Abbauen von Zellen....27
5.2.3.1 Die Lebenskraft....28
5.2.3.2 Absturz durch Fehler....29
5.3 Der Regenerierungseffekt....30

6 DNA-Der Programmcode der Zelle 32

7 Immunsystem-Sicherheit im System....33
7.1 Erkennen von Fremdzellen....33

7.2 Mobile Sicherheitszellen....34

8 Kommunikation zwischen den Zellen....36
8.1 Direct Communication System DCS....36
8.1.1 Steuern von Zellen durch eine Zentrale....36
8.1.2 Workflow....37
8.2 Open Communication System OCS....38
8.2.1 Das OCS nach dem Pool-Prinzip....39
8.3 Beispiel: Hacker Angriff....40

9 Arbeitsweise der Zellen....41
9.1 Aufteilen der Arbeit....42
9.2 Beschreiben eines Workflows....43
9.2.1 Definieren von Bedingungen....44
9.2.2 Synchronisieren....45
9.2.2.1 Das Synchronisationsverfahren....46
9.2.3 Parallelisieren von Abläufen....48
9.2.4 Definieren von Workflows....49
9.3 Beispiel Versorgung mit Aufträgen....50

10 Kommandozentrale des Systems....51
10.1 Steuern der Zellen....51
10.2 System Erfahrung....52
10.3 Evolutionäre Weiterentwicklung....52

11 Die Datenablage....53
11.1 Lokale Zellendaten....53
11.2 Die Speicherzellen....53
11.3 Beispiel einer Datenverwaltung....54
11.4 Einige Zellenarten....54

12 Einleitung zum Framework....57
12.1 Das System....57
12.2 Einsatzgebiet....58
12.2.1 Clustering....58
12.2.2 Bediensoftware für Maschinen....60
12.3 Vorgehen....61

13 Module Object Model Spezifikation 2.0....62
13.1 Eigenschaften des MOM....62
13.2 Aufbau und Definition....63
13.2.1 Basis Anforderungen....63
13.2.2 Erweiterungen....64

14 Das Klassen-Framework....66
14.1 Vorschriften bei der Entwicklung mit Zeus....67
14.1.1 Schnittstellen....67
14.1.2 Die Basisschnittstelle IZUnknown....69
14.1.3 Die Basisklasse TZObject....71

14.2 Datentypen des Zeus-Frameworks....73
14.3 Klassen zum MOM 2....74
14.3.1 Implementation eines X-Objekts....77
14.3.2 Wichtige Methoden des X-Objekts....80
14.3.3 Objektpfad....81
14.4 Die Zellenarchitektur....82
14.4.1 Die Zellen....82
14.4.2 Threading....84
14.4.3 Networking....85
14.4.4 Cell Communication Transfer Protocol....85
14.4.4.1 Request Pakete....86
14.4.4.2 Response Pakete....87
14.4.5 Messaging....87
14.4.5.1 Die Kontrollmeldungen....88
14.4.5.2 Die Arbeitspakete....89
14.4.6 Kommunikationskanäle....89
14.4.6.1 Lokale Kommunikationskanäle....92
14.4.6.2 Entfernte Kommunikationskanäle....93
14.5 Serialisierung von Objekten....95
14.5.1 Streaming....100
14.6 Remote Method Invocation mit C++....100
14.7 Klassen für künstliche Intelligenz....101

15 Definition von X-Objekten in XML....102
15.1 Allgemeine XML Spezifikation....102
15.1.1 Objekte....102
15.1.2 Attribute....103
15.2 Erweiterte Definition zum ClassLoader....105

16 Entwickeln von Code-Modulen....106
16.1 Code Modul zur Erzeugung von Objekten....107
16.2 Code Modul zum Erstellen einer Session....107
16.3 Beispiel einer Code-Modul Implementation....110

17 Das Sicherheitskonzept....113
17.1 Sicheres Laden von Code-Modulen....113
17.2 Überwachung der Zellen....114

18 Einleitung Projekthandbuch....116

19 Projektbeschreibung....117
19.1 Ausgangssituation....117
19.1.1 Bestehende Produkte zu CCM....118
19.2 Problemstellung....118
19.2.1 Spezifikation zum Cell Computing Model....118
19.2.2 Implementation der Finger Print Technik....119
19.2.3 Beispiel Implementation....119
19.2.4 Reflektieren und Verbessern des System....119

19.2.5 Mobile Zellen (fakultativ)....120
19.3 Projektrisiken....120
19.4 Ziele....122
19.5 Vorgehensstrategie....122

20 Projektspezifisches Vorgehensmodell....123

21 Entscheidungspunkte und auszuliefernde Ergebnisse....124

22 Methoden und Werkzeuge....125
22.1 Projektmanagement PM....125
22.2 Software Engineering SE....125

23 Standards und Richtlinien....126

24 Ergänzende Projektvereinbarungen....127
24.1 Projektorganisation....127
24.2 Projektplanung....128
24.3 Aufwandschätzung....128
24.3.1 Arbeitspaket: Beispiel Bildverarbeitungsprozess....130
24.3.2 Arbeitspaket: Sicherheitskonzept für CCM....132

25 Einleitung Schlussbericht....134

26 Soll-Ist Vergleich....135
26.1 Abweichungen zu den geplanten Termine und Ergebnissen....136

27 Erkannte (ungelöste) Probleme 137
27.1 Technische Probleme....137

28 Erreichte Ziele....138
28.1 Das Zeus-Framework....138
28.1.1 Einfachheit....140
28.1.2 Schnelligkeit....140
28.1.3 Performance....140

A Applikationen des Framework Zeus....141
A.1 Bibliotheken....141
A.2 Code-Module....142
A.3 Programme und Skripts....142
A.4 Benutzerhandbuch zum Framework [zeus]....143
A.4.1 Konfiguration....143
A.4.2 Kommandozeilen Parameter....145
A.5 Der Namensdienst....146
A.5.1 Konfiguration der Zellen....147

B Messungen vom Klonverhalten....148
B.1 Klonen von Zellen....148
B.2 Abbauen von Zellen....148
B.3 Messungen....149

C Messungen mit Shared Memory.....152
C.1 Konfiguration.....153

D Messungen mit verteilten Applikationen.....155
D.1 Unterschied der Performance (Inprozess oder Verteilen).....155
D.2 Vergleichsmessung.....156
D.3 Konfiguration.....158

E Versionsgeschichte.....161

Literaturverzeichnis.....162

1 Vorwort

Durch den steigenden Bedarf an Informationen und durch die globale Vernetzung der Gesellschaft werden Softwarelösungen komplexer und dadurch schwerer wartbar. Wie sehen Softwarelösungen zukünftig aus, und wo haben wir noch Potential bei der Verarbeitung von Informationen? In diesem Zusammenhang wurde das Cell Computing Model entworfen. Es definiert ein Software Modell, welches Aufbau und Arbeitsweise von biologischen Organismen zum Vorbild nimmt. Durch die Kombination mit bewährten Paradigmen der objektorientierten Entwicklung wurde ein Framework geschaffen, welches die Entwicklung und Anwendung von Cell Computing Applikationen ermöglicht.

In der realen Welt laufen unendlich viele Ereignisse zur gleichen Zeit ab, unzählige Kreisläufe stabilisieren das System in stetiger Wechselwirkung. Durch die Evolution werden unflexible und störende Komponenten zerstört oder angepasst. Dieses Verhalten wird mit der Definition des Cell Computing Model (CCM) in die Computerwelt übertragen, wobei sich dieses Modell sehr stark an die biologische Arbeitsweise und den Aufbau von Lebewesen orientiert. Das Modell richtet sich nach Multiprozessor Computer und verteilten Applikationen aus. In absehbarer Zukunft stösst die Entwicklung von noch schnelleren Rechnern an ihre Grenzen. Die Vernetzung und der Bedarf an Information wird aber stetig zunehmen. Dadurch wird die Entwicklung von Applikationen komplexer und aufwendiger.

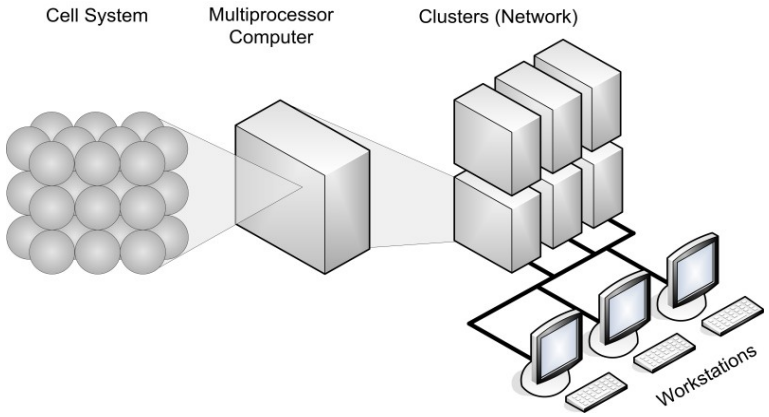


Abbildung 1: Verteilte Applikation mit Cell Computing Model

Im Rahmen dieser Diplomarbeit wurde das CCM definiert und ein Framework geschaffen, welches das Entwickeln von komplexen und verteilten Applikationen auf Basis des CCM ermöglicht. Wie der Name Cell Computing

bereits besagt, werden so genannte Programmzellen entwickelt, die eine bestimmte Aufgabe im System übernehmen sollen. Diese Zellen laufen parallel und kommunizieren miteinander. Sie tauschen Daten untereinander aus. Zellen sterben ab, wenn sie nicht mehr gebraucht werden, und es werden je nach Bedarf wieder neue generiert. Dadurch passt sich ein System stets der Aufgabe an und es kann sich sogar weiterentwickeln. Neue Zellen können von der Erfahrung bestehender Zellen profitieren.

Die Systemsicherheit wird durch mobile Sicherheitszellen realisiert. Sie wandern im System und siedeln sich dort an, wo sie dringend gebraucht werden. Sie blockieren und neutralisieren Fremdzellen, zum Beispiel Viren.

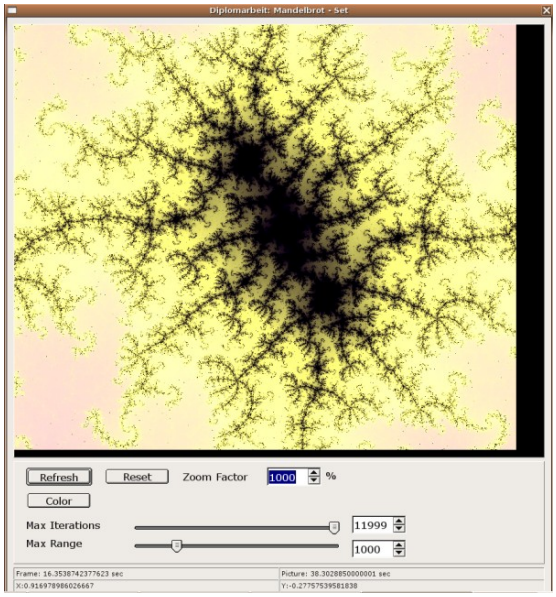


Abbildung 2: Beispiel zum Berechnen von Fraktalen

Noch bleibt viel zu tun, denn die Hardware von heute ist noch nicht für diese extrem parallele Verarbeitung gebaut. Dennoch kann dieses Modell bereits mit mehreren vernetzten PC's eingesetzt werden. Mehr Informationen zur Entwicklung und den Möglichkeiten sind unter http://www.xatlantis.ch zu finden.

2 Einleitung

Seit dem Beginn der Computer wurden immer wieder neue Technologien und Methoden entwickelt, um gewisse Probleme einfacher und strukturierter zu lösen. Mit der Maschinensprache wurden die ersten Programme für Computer geschrieben. Die darauf folgenden Hochsprachen waren einfacher zu lesen und anzuwenden, die Programmierung erfolgte aber nach dem gleichen Prinzip, nämlich linear (Prozedural). Es entstanden Probleme mit der Strukturierung von umfangreicherer Software. Die Lesbarkeit und dadurch die Wartung der Software waren schwierig und kompliziert. Mit der objektorientierten Programmierung (OO-Programming) wurde eine „neue“ Denkweise ins Leben gerufen. Die Modellierung der Software wurde an die reale Welt angepasst. Objekte aus der realen Welt werden abstrahiert und in die Entwicklung der Software miteinbezogen, sie werden also abgebildet. Diese Objekte können miteinander kommunizieren und Daten austauschen.

In den letzten Jahren ist der Umfang der Software und deren Anforderungen ständig gestiegen. Mit der weltweiten Vernetzung, dem Internet, wurde die Entwicklung von verteilten Systemen (zum Bsp. Web-Services, Java) stark gefördert. Die Objekte befinden sich nicht mehr nur auf einem Computer, sondern existieren dezentral und kommunizieren über das Netzwerk miteinander. Das Ziel ist, in einem Netzwerk von Computer verschiedene Dienste bereitzustellen, welche von anderen Computer oder Computersystemen gebraucht werden können.

Was die Zukunft bringt weiss niemand. Aber der heutige Trend in der Computerwelt zeigt, dass die Informationsmenge nicht geringer wird. Im Gegenteil. Die Verarbeitung der Informationen wird immer aufwendiger. Bilder können mit hoch genauerer Auflösung erstellt und gespeichert werden, Filme werden mit hoher Qualität über Netzwerke verteilt und umgewandelt. Bald wird das Fernsehen live und mehrkanalig auch über das Internet übertragen. Aber auch im Bereich der Geschäftswelt wird die Datenmenge grösser. Die Vernetzung der Gesellschaft spiegelt sich auch in den Systemen wieder, welche die Gesellschaft braucht, um ihre Aufgaben zu erledigen. Dazu werden in der Geschäftswelt immer kürzere Antwortzeiten verlangt. Das erfordert einerseits schnelle und anderseits flexible Systeme. Der Wunsch nach Systemen, die sich autonom regenerieren und sich an der Umgebung und der Aufgabe anpassen, wächst. Heute werden diese Systeme mit dem objektorientierten Ansatz entwickelt. Durch das Anwenden von Design Patterns und anderen Implementationstechniken wird versucht schneller und durchgängiger zu entwickeln. Die Entwicklungszeit solcher Systeme ist jedoch relativ lang und deshalb auch teurer. Die finanziellen Ressourcen der Käufer ist mehr oder weniger begrenzt, und so ist es auch verständlich, dass solche Systeme in der Geschäftswelt nicht gerade zu einer Euphorie führen.

3 Was ist Cell Computing Model?

Es stellt sich die Frage, ob nicht ein neues dem OO-Programming übergeordnetes Konzept geschaffen werden könnte, um die genannten Probleme und Anforderungen an zukünftige Systeme besser in den Griff zu bekommen. Wie könnte ein solches System aussehen?

Es gibt in der realen Welt einen riesigen Unterschied zu den heutigen Computersystemen: In der realen Welt laufen unendliche Ereignisse (Tasks) parallel ab, Arbeiten und Abläufe geschehen gleichzeitig. Es existieren unzählige Kreisläufe, die das System stabilisieren und ausbalancieren. In der Computerwelt stellt sich nun langsam die Frage, ob solche Verhaltensweisen nicht auch erwünscht wären. Die Datenmenge artet langsam in eine richtige Flut aus, das Verarbeiten wird dadurch nicht kürzer. Auch mit den immer leistungsfähigeren Mikroprozessoren wird einmal Schluss sein. Hier stösst die Technik heute schon an gewisse Grenzen. Doch die Vernetzung und Verarbeitung wird mit dem Wachstum unserer Gesellschaft stetig zunehmen, vielleicht sogar exponentiell. Dieses Dokument zeigt eine Möglichkeit auf, wie zukünftige Software geschrieben werden kann, nämlich mit den Erkenntnissen der Genforschung.

Die Natur könnte uns also auch hier wieder einmal als Vorbild dienen, um solche Systeme zu entwickeln. Das folgende Unterkapitel befasst sich kurz mit der realen Welt und welche Prinzipien zum Bau des Systems gebraucht werden können.

3.1 Die Welt der chaotischen Systeme

Die Erde besteht aus 3 Hauptkreisläufen, der Atmosphäre, den Meeresströmungen und der Magmaströmungen. Diese Kreisläufe sind chaotisch organisiert und beeinflussen sich gegenseitig. Sie geben dem System Erde die Stabilität, welche es schlussendlich ermöglicht, dass Lebewesen existieren können. Die Lebewesen ihrerseits helfen aber auch mit, das System zu steuern und zu stabilisieren. Sie sind in diese Kreisläufen integriert. Wird ein Kreislauf instabil, wird er durch andere Systemkomponenten wieder stabilisiert. Dabei ist das Prinzip der Evolution nicht irrelevant. Komponenten, die sich störend entwickeln, werden zerstört oder mit der Zeit verändert und an die Umgebung angepasst. Dieses Prinzip hat sich über mehrere Milliarden Jahre bewährt.

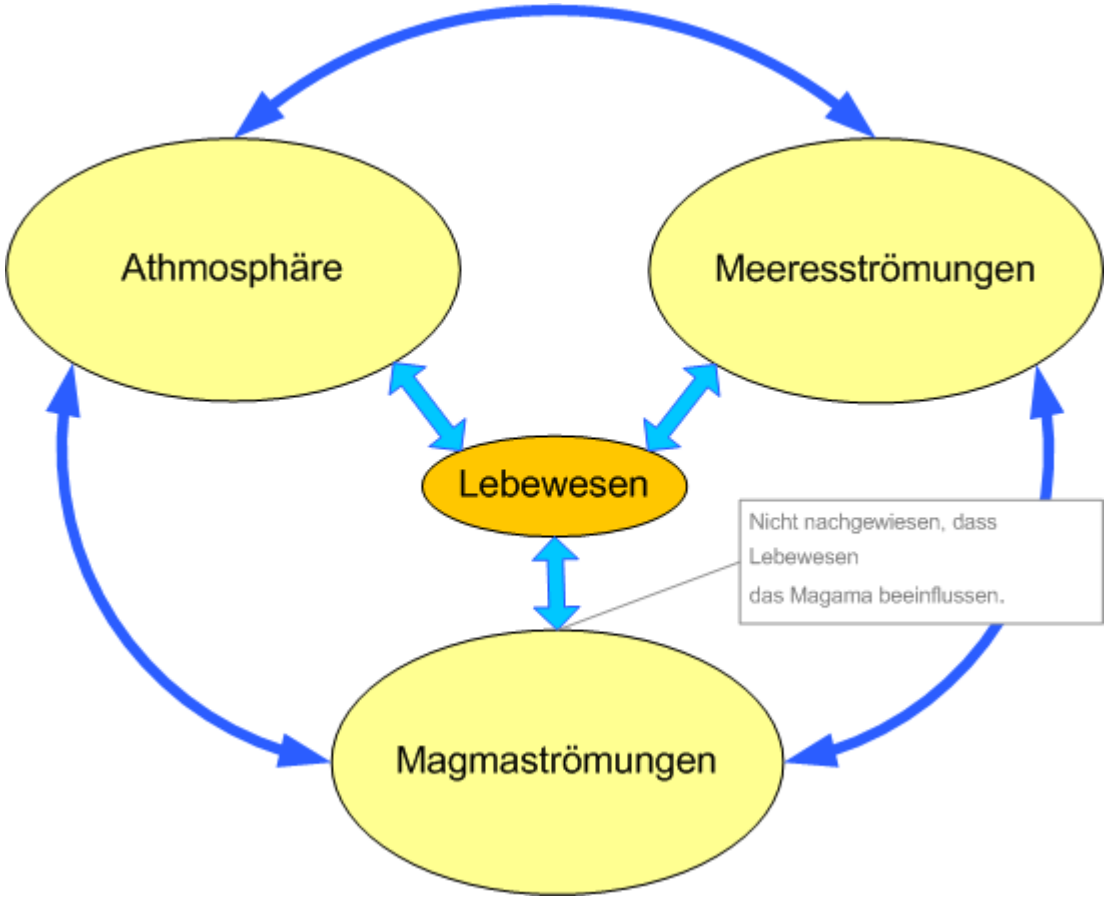


Abbildung 3: Die 3 chaotischen Systeme der Erde, die das Leben beeinflussen, und welche vom Leben beeinflusst werden.

Beim Betrachten eines Lebewesens, wird klar, dass sich auch die internen Systeme eines jeden Lebewesens ähnlich verhaltet wie das ganze System Erde. Krankheiten werden bekämpft, das Immunsystem aufgebaut und weiter entwickelt.

3.1.1 Aufbau der Lebewesen

Die Lebewesen auf der Erde sind aus Zellen aufgebaut. Die Zellen stellen sozusagen kleine Systeme dar, die mit der Aussenwelt und ihren benachbarten Zellen interagieren. Diese Interaktion ist für die Zelle lebenswichtig. Sie können also nicht als eigenständig bezeichnet werden. Sie brauchen die Umgebung, um zu existieren. Aber auch die Umgebung ist von den Zellen abhängig. Die Zelle ist fest in einem oder mehreren der 3 Kreisläufe beteiligt.

Je nach Komplexität des Lebewesen gibt es unterschiedliche Zellenverbunde (Zellenhaufen). Die Zellenverbunde bestehen aus einem bestimmten Zellentyp, der dem Lebewesen einen bestimmten Dienst erweist.[1]

3.1.1.1 Stammzellen

Interessant ist das jedes Lebewesen aus einer einzigen Zelle entsteht. Diese Zelle teilt sich nun, es entsteht ein Zellenhaufen und schlussendlich das vollständige Lebewesen.

Stammzellen sind die Bausteine des Lebens (Urzellen). Sie können sich fast unbegrenzt teilen. Aus ihnen entwickeln sich alle anderen Zellen eines Lebewesens. Die ersten Zellen sind die embryonale Stammzellen. Aber auch nach der vollständigen Entwicklung des Lebewesens existieren Stammzellen, um den Körper zu heilen und zu reparieren.

3.1.1.2 Spezialisierte Zellen

Aus den Stammzellen entstehen spezialisierte Zellen. Diese nehmen nun eine bestimmte Aufgabe wahr. Sie sind die Bausteine der Organe (Haut, Leber etc.), der Nerven, Muskeln und des Bluts. Je nachdem, wie gross und umfangreich die einzelnen Aufgaben sind, existieren mehr oder weniger Zellen einer Zellenart. Gewisse Arten werden erst in grösserem Ausmass produziert, wenn sie dringend gebraucht werden. Andere sind einfach als Reservezellen vorhanden.

3.1.2 Generierung und Regenerierung

Ein sehr wichtiges Konzept der Natur besteht darin, abgestorbene Zellen

1 Wissen über die biologischen Zusammenhänge stammen aus [GEOK2] und [MSENC]

durch neue zu ersetzen. Stirbt eine Zelle ab, übernimmt die Aufgabe eine benachbarte Zelle. Laufend werden die abgestorbenen Zellen durch neue, herangewachsene Zellen ersetzt. Dieser Regenerierungsprozess wird vor allem bei Zellen eingesetzt, die unter hoher Belastung stehen oder für das System von grosser Bedeutung sind.

3.1.3 Das Programm der Zelle (DNA)

Wie eine Zelle arbeiten soll, steht in der so genannten DNA (deoxyribonucleic acid), dem Programm der Zelle. Mit Hilfe der DNA werden Proteine produziert, die dann die Aufgaben der Zelle übernehmen, wie Stoffwechsel, Produzieren von anderen Zellen usw. Eine Leberzelle braucht also nicht denselben DNA Abschnitt wie eine Hautzelle. Dennoch ist die ganze DNA in jeder Zelle vorhanden. Die Zelle braucht aber nur einen bestimmten Ausschnitt aus dem DNA für ihre Arbeit. Die nicht verwendeten Abschnitte der DNA sind komprimiert. Die Stammzellen (Siehe Stammzellen) sind noch nicht spezialisierte Zellen, das heisst, sie können einen grösseren Abschnitt der DNA verwenden und mehrere Arbeiten erledigen. Durch das Teilen der Stammzellen entstehen dann immer spezialisiertere Zellen, deren DNA-Abschnitt gegenüber der Stammzelle eingeschränkter ist.

3.1.3.1 Wechseln des DNA Abschnitts

Zellen können auch den DNA Abschnitt wechseln. Somit kann sie bei Bedarf helfen, andere Aufgaben zu erledigen. Geschieht dies aber unkoordiniert, so sprechen wir etwa von Krebs oder einem Geschwür.

3.1.4 Das Immunsystem

Die Sicherheit oder Lebensfähigkeit bezüglich Krankheiten wird bei komplexeren Lebewesen durch das Immunsystem realisiert. Das geschieht durch Antikörper und den weissen Blutzellen, die eine Art Polizei bilden. Krankheiten werden durch das Immunsystem erkannt. Als Reaktion werden Antikörper produziert, welche die Erreger neutralisieren. Fremdkörper und neutralisierte Erreger werden durch die weissen Blutzellen gefressen und somit entfernt.

3.2 Ideenkatalog

Das neue Modell soll nach dem Grundsatz der Natur gebaut werden. Es existiert aus Zellen, was ein modularer Aufbau bedeutet. Die Struktur gibt auch hier dem Modell den Namen: Computing Cell Model CCM. Das CCM soll gewissermassen einem lebenden Organismus entsprechen.

Die wichtigsten Merkmale von CCM sollen hier stichwortartig aufgeführt sein:

- **Regenerierung**.
  - Das Generieren und Teilen von Zellen, das Wegräumen der abgestorben Zellen (Verwaltung der Ressourcen)
  - Erkennen von Fremdkörper und deren Bekämpfung (Security)

- **Parallele Verarbeitung** der Aufgaben (Multitasking)
  - Aufteilen, Verarbeiten und Zusammensetzen der Aufgaben
  - Autonome Anpassung an die Umgebung und an die Aufgabe (KI)

- **Lernfähigkeit**. Das System soll gewisse Abläufe selbständig optimieren, Erfahrungen sammeln und Verhaltensmuster lernen können (neuronal evolutionäres Prinzip).

3.2.1 Regenerierung

Ein sehr wichtiges Ziel ist die Stabilität und Systemsicherheit. Mit der Regenerierung von Zellen und einer geschickten Arbeitsteilung soll ein möglichst absturzsicheres Konzept realisiert werden. Mit automatischem Erkennen von Fehler und Einleiten der entsprechenden Massnahmen soll sich das System selber wieder stabilisieren können. Sterben Zellen ab (durch Fehler oder durch Angriffe von Fremdkörper), übernimmt die Arbeit eine benachbarte Zelle.

Zudem sollen Angriffe auf das System von Aussen besser erkannt werden können. Im weitesten Sinne würde dies einer Antivirus-Software entsprechen. Das System erkennt Viren automatisch und beseitigt sie durch entsprechende Gegenmassnahmen. Durch die Fähigkeit Erfahrungen sammeln zu können, soll das System gegen Virenarten immun werden.

3.2.2 Parallele Verarbeitung der Daten

Das System soll durch parallele Verarbeitung riesige Datenmengen schneller verarbeiten und analysieren können.

Die Performance des Systems soll sich nach dem Arbeitsvolumen richten. Stehen viele Aufträge an, vermehren sich gewisse Zellenarten, um der grösseren Arbeitslast entgegen zu wirken. Dagegen schrumpft das System, wenn wenig Arbeit zu verrichten ist (ähnlich wie beim Muskel). Dieses Auf- und Abbauen geschieht nicht unmittelbar, sondern ist eine träge Angelegenheit. Dieser Vorgang kann durch andere Systemkomponenten auch beeinflusst werden.

3.2.3 Lernfähigkeit

Das System soll, gegenüber heutiger Systeme, autonomer werden. Das heisst, das System soll sich den Anforderungen dynamisch anpassen können, Erfahrungen sammeln und diese Erfahrungen anwenden können. Damit soll die Lebensfähigkeit und damit auch die Lebensdauer des Systems deutlich erhöht werden. In einer ohnehin schon sehr kurzlebigen Zeit könnte so gerade in der Informatik etwas nachhaltiger entwickelt werden.

Das CCM wird durch ein neuronales Netzwerk erweitert. Dieses Netzwerk dient als Gehirn (Kommandozentrale) des Systems. Erfahrungen und verarbeitungstechnische Daten werden hier miteinander verknüpft. Das CCM wird so zu einem lernfähigen, autonomen System.

3.3 Zieldefinition

Das Cell Computing Model ist ein Konzept, welches auf dem objektorientierten Modell aufbaut. In folgenden Bereichen soll eine Vereinfachung gegenüber dem objektorientierten Modell erzielt werden:

- Schnellere Entwicklung von autonomen, lernfähigen Systemen
- Entwicklung von komplexen Systemen vereinheitlichen
- Stabile Systeme bezüglich Absturz und Angriffe von aussen
- Parallel processing für Datenverarbeitung

- Verteiltes System, welches die heutigen Techniken wie Webservices unterstützt und erweitert (XML, SOAP).
- Verwenden gängiger Netzwerke und des Internets (TCP/IP)
- Framework für künstliche Intelligenz.

3.3.1 Framework

Mit dem CCM muss ein Framework entwickelt werden, welches erlaubt, Zellen und deren DNA zu programmieren. Das Framework ist eine Software, welches dem Programmierer und Entwickler die grundsätzlichen Strukturen und Rahmenbedingungen vorgibt, und somit die Entwicklungszeit reduziert.

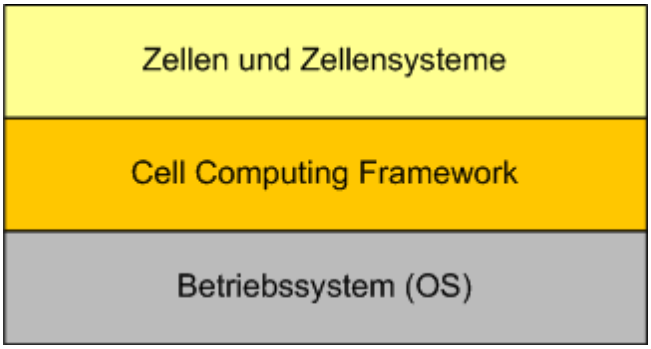


Abbildung 4: Einbettung in ein Betriebssystem

Das Framework setzt auf einem Betriebssystem auf, welches mindestens multitasking fähig und netzwerkfähig ist.

Aufgaben des Frameworks:

- Gemeinsame Basis für Kommunikation
- Verwaltung der Ressourcen
- API (Application Program Interface) für künstliche Intelligenz
- API für Zellen Programmierung (DNA)
- ev. Windows Manager (Graphische Oberfläche)

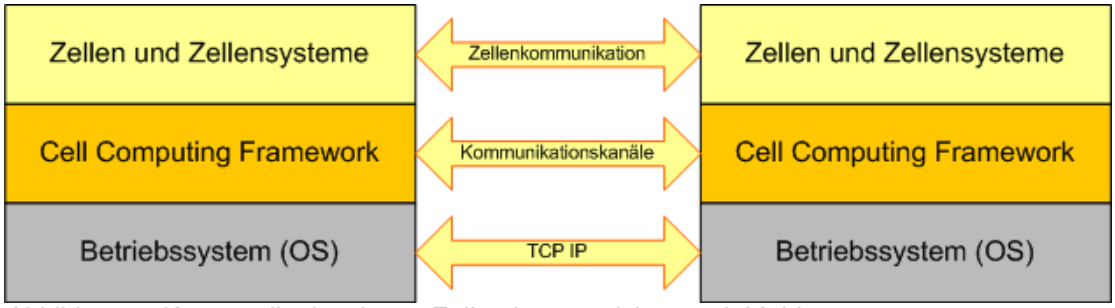


Abbildung 5: Kommunikationslayer. Zellen kommunizieren mit Meldungen.

3.3.2 Hardware

Eine optimale Hardware für das CCM ist ein Multiprozessor-System. Jede Zelle entspricht mindestens einem Task im herkömmlichen Sinn. Will dieser Task ohne Context Switching arbeiten, also real Multitasking, muss jede Zelle ihren eigenen Prozessor haben. Das heisst es existieren Tausende, vielleicht Millionen von Prozessoren in diesem Hardwaresystem.

Dies kann bereits mit herkömmlichen Netzwerken erreicht werden. Mehrere Computer kommunizieren über dieses Netzwerk miteinander. Das CCM wird also als ein verteiltes System betrieben.

In den Anfängen der Entwicklung des CCM kann auch mit einem Einzelprozessor gearbeitet werden. Gewisse Verhaltensweise können so bereits Umgesetzt werden. Zudem sind die Einzelprozessoren relativ kostengünstig.

3.4 Einsatzgebiet

3.4.1 Vorteile

Vorteile sind immer relativ zu den existierenden Systemen auszumachen. Ein wichtiger Punkt ist sicher die Autonomie des Systems. Es kann sich selbstständig an der Umgebung und an die Aufträge anpassen, was eine sehr hohe Flexibilität und Lebensdauer mit sich bringt. Durch künstliche Intelligenz und der Lernfähigkeit kann das System gewisse Entscheidungen selber treffen und Abläufe noch optimieren.

Die Vermehrung jener Zellen, welche sehr oft und intensiv genutzt werden, erhöht die Performance des Systems (Sofern mehrere Prozessoren zur Verfügung stehen). Somit kann das System je nach Arbeit reagieren und sich anpassen. Durch so genannte Sicherheitszellen wird das gleiche Schema für die Systemsicherheit implementiert. Je mehr das System angegriffen wird (durch Hacker, Fremdzellen usw.), desto mehr Sicherheitszellen werden produziert. Zudem kann das System gegen gewisse Fremdzellen immun werden, indem gewisse Zellenarten von Anfang an als schädlich erkannt und somit neutralisiert werden können.

Ein weiterer Vorteil liegt in der hohen Stabilität des Systems. Es können immer nur kleine Teile dieses System zerstört werden, und diese werden fortlaufend ersetzt. Es handelt sich hier um einen Regenerierungsseffekt.

3.4.2 Nachteile

Das System braucht enorme Hardware in Bezug auf Schnelligkeit und Prozessorleistung. Man kann sich etwa vorstellen, dass jede Zelle ein Thread oder sogar ein eigener Prozess im heutigen Sinne der Informatik ist. Bei herkömmlichen Systemen (PC mit einem Prozessor) wird mit context switching gearbeitet (Pseudo multi threading). Dies bringt nicht eine wesentliche Steigerung der Leistung. Im Gegenteil: bei gewissen Aufträgen verliert man Leistung durch den Wechsel von einem Thread zum anderen (Scheduling). Um eine grosse Leistungssteigerung vollbringen zu können, sollte jede Zelle einen eigenen Prozessor (Hardware) für sich beanspruchen können. Das heisst für die Hardware, dass CCM einen Parallelrechner braucht. In dieser Hinsicht ist der Erfolg von CCM von zukünftigen Hardwareentwicklungen abhängig (Siehe Zukunftsvisionen).

Bei sequenziellen Systemen kann zu jedem Zeitpunkt genau gesagt werden, welches der nächste Verarbeitungsschritt ist. Auch bei der OOP ist genau definiert, welches Objekt mit welchem Objekt wie kommuniziert. Im CCM ist dies nicht mehr eindeutig feststellbar. Welche Zelle nun gerade welchen Auftrag verarbeitet, kann nicht vorausgesagt werden. Auch wie viel das System wächst bzw. schrumpft kann höchstens geschätzt werden. Es kann höchstens ein Trend festgestellt werden. Wie beim Wetter, bei welchem man auch nie hundertprozentig vorhersagen kann, wie der morgige Tag aussieht. Dieses Verhalten könnte als ein Nachteil empfunden werden.

3.5 Zukunftsvisionen

Wie sich die Computertechnologie in mittlerer Zukunft entwickelt, lässt sich nicht so leicht analysieren. Sicher ist, dass mit der neuen Quanten- und Nanotechnologie ein gewaltiger Schritt in Richtung Kleinstprozessoren gemacht wird. Eine meiner Visionen ist, dass sich in Zukunft ein PC nicht nur aus einem Einzelprozessor aufbaut, sondern dass man, ähnlich wie beim RAM-Speicher, Processor Banks kaufen kann. Auf solchen Banks befinden sich unzählige Prozessoren, die von einem Programm (oder eine Zelle) aktiviert werden, und nun die Aufgabe dieses Programms bewältigen. Ähnlich wie ein Speicherbaustein kann nun eine Zelle einen freien Prozessor aktivieren. Werden Prozessoren nicht gebraucht, befinden sie sich im Leerlauf. Diese Technologie würde es dann erlauben, eine Menge von Prozessen in Echtheit parallel arbeiten zu lassen. Das CCM würde sich hervorragend für solche Hardware eignen, da es für das parallele Abarbeiten von Aufträgen konzipiert ist.

Zukünftig sollen alle Computersysteme mit CCM ausgestattet werden. Zusammen mit der Intelligenz des Erfahrungssystems (Gedächtnis des Systems) könnte das System die neue Computersoftwarerevolution bedeuten. Computer nehmen dem Menschen nicht nur rechnerische und datenverwaltende Aufgaben ab, sondern nun auch das Treffen von minderwertigen Entscheidungen. Sie werden auch hier nicht, wie schon vor etlichen Jahren befürchtet, den Menschen ersetzen, sondern ihm lästige oder zeitkritische Arbeit und Entscheidungen abnehmen. Lernfähige Systeme werden in der Wirtschaft wie auch für militärische Zwecke gebraucht, um aus Erfahrungen und neuen Gegebenheiten Entscheidungen treffen zu können.

4 Grundkonzept des CCM

Der folgende Abschnitt zeigt eine mögliche Umsetzung des Cell Computing Model auf. Dabei wird immer wieder auf die Synergien zur realen Welt zurückgegriffen. Gerade dieser Abschnitt kann sich im Verlaufe der Entwicklung noch stark verändern, da viele Strategien zuerst noch getestet und erforscht werden müssen. Es ist auch nicht ausgeschlossen, dass neue Erkenntnisse aus der Genforschung die Entwicklung dieses Projekts beeinflussen werden. Deshalb kann es sich hier nicht um eine endgültige Lösung handeln. Vielmehr soll es zum Fördern von Ideen und neuen Ansätzen dienen.

4.1 Das EVA-Prinzip

Die Grundidee des Informatiksystems bleibt auch hier bestehen. Es handelt sich also ebenfalls um ein EVA-System (Eingabe-Verarbeitung-Ausgabe). Das CCM Konzept beinhaltet eine Dateneingabe, Datenverarbeitung und eine Datenausgabe. Der grosse Unterschied ist aber in der Systemarchitektur und der Art der Datenverarbeitung zu finden. Im Weiteren wird die Peripherie anders benannt. So heissen die Eingabegeräte mit dem entsprechenden Zellenhaufen **Sensoren** (Kamera, Maus, Tastatur etc.), die Ausgabegeräte werden als **Aktoren** bezeichnet (Drucker, Bildschirm, Roboterarm etc.). Diese Bezeichnungen sollen abstrakt verstanden werden.

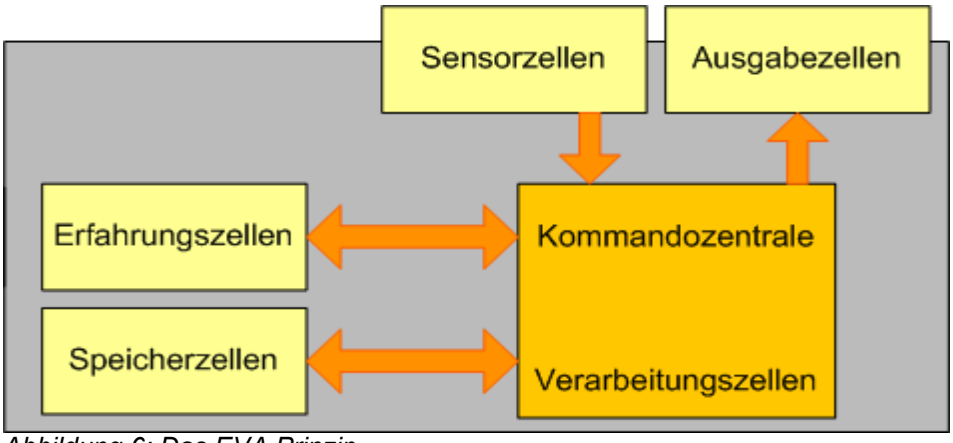


Abbildung 6: Das EVA Prinzip

Es können sowohl virtuelle wie auch reale Geräte sein. Durch die Sensoren kann das System zu Daten gelangen, während bei Aktoren irgend ein Resultat der Eingabedaten und deren Verarbeitung sichtbar wird, zum Bsp. Bewegung eines Roboters.

Das Prinzip wird mit Hilfe von Tastaturen und Bildschirmen verdeutlicht:

An einem solchen Grosssystem können zum Beispiel hunderte Tastaturen vorhanden sein. Jede spricht eine Sensorzelle an. Diese Zellen nehmen die Eingaben entgegen und speisen damit das System. Jeder Bildschirm wird von einer oder mehreren Ausgabezellen angesprochen. Das System kann nun Informationen und Daten über diese Ausgabezellen an den Bildschirm weitergeben. Dabei könnte es sich auch um hunderte von Bildschirmen handeln.

Wo diese Zelle lebt kann je nach Hardware verschieden sein. Sie könnte zum Beispiel direkt in der Tastatur existieren. Das hätte den Vorteil, dass der Treiber sich bereits bei der richtigen Hardware befindet. Kurz gesagt: die Software und Hardware verschmelzen zu einer Einheit.

Die Steuerung des Systems erfolgt durch eine oder mehrere Kommandozentrale(n). Die erlernten Erfahrungswerten werden hier verknüpft und verwaltet. Durch Kommunikationssysteme kann die Zentrale die einzelnen Zellenhaufen steuern und so Einfluss auf den Verarbeitungsprozess nehmen. Damit kann die Möglichkeit einer autonomen Systemverbesserung und Optimierung erzielt werden, was wiederum zu einer besseren Lebensfähigkeit führt. Ein weiterer Ansatz zur Prozessoptimierung ist das Züchten von guten und das Zerstören der weniger effektiven Zellen (evolutionäres Prinzip).

Wie sieht es bei der Datenablage aus? Die Datenablage ist einem Datenarchiv ähnlich. Durch Verzeichnisse kann der Benutzer Strukturen anlegen und darin seine Dokumente speichern. Hier wird das CCM keine Änderung bringen. Im Gegenteil sollen hier die vorhandenen Technologien weiter verwendet werden.

Irgendwo müssen die Zellen ihre Arbeiten holen bzw. speichern, Daten im RAM zwischenspeichern und Erfahrungen verknüpfen. Aber auch hier können wir vorhandene Technologien verwenden, die sich gerade in der objektorientierten Programmierung bewährt haben. Im CCM unterscheiden wir folgende Datenarten:

- **Zellendaten**. Diese Daten verwendet eine Zellen nur intern. Sie sind die Daten der Zellenattribute. Diese Daten sind von aussen nicht zugänglich.

- **Speicherdaten**. Diese Daten entsprechen sozusagen den gespeicherten

Dokumenten und der verarbeitete Daten des Benutzers.

- **Erfahrungsdaten** sind Einstellungen und Erfahrungen des Systems. Diese Daten werden von der Kommandozentrale verwaltet. Diese Daten könnten zum Beispiel in einem neuronalen Netz verknüpft gespeichert werden.

5 Das Zellenleben

Dieser Abschnitt stellt zwei Möglichkeiten vor, Zellen zu generieren. Bei der ersten Variante werden die Zellen durch eine Zellenfabrik erzeugt. Bei der zweiten Variante wird das Klonen angewandt. Beide Varianten werden auch von der Natur eingesetzt, um Zellen zu produzieren. Dabei ist entscheidend, wo und wie die Zellen eingesetzt werden, und damit natürlich um welche Zellenart es sich handelt.

5.1 Der Lebenszyklus

Jede Zelle durchlebt einen Zyklus. Ein Zyklus umfasst folgende Stadien:

- Zelle wird generiert
- Zelle ist inaktiv (dient als Reserve)
- Zelle wird aktiviert und kann nun Daten verarbeiten
- Zelle wird wieder inaktiv wenn keine Daten vorhanden sind
- Zelle stirbt ab, wenn sie
  - Fehlerhaft ist
  - Gefährlich ist
  - Zu lange inaktiv ist (verhungert)

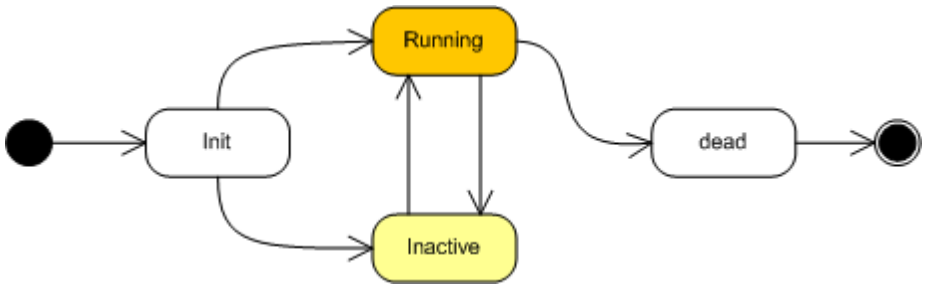


Abbildung 7: Zustände einer Zelle

In einem multitaskfähigen Betriebssystem können Zellen als Prozesse oder Threads implementiert werden.

5.2 Erzeugen und Abbauen von Zellen

Das System soll sich dynamisch bezüglich der Eingangslast verhalten. Dazu muss ein Verfahren gefunden werden, welches sich möglichst stabil verhält.

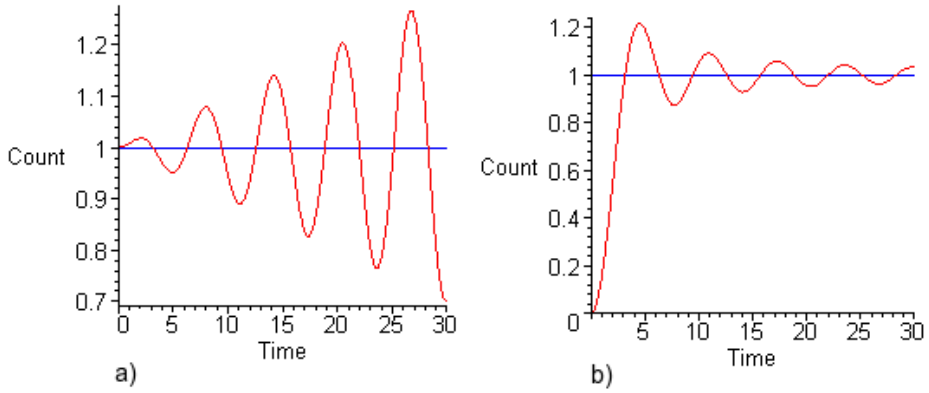


Abbildung 8: Zeigt die Verhaltensweisen von Klonalgorithmen welche mit der Zeit instabil werden (a) oder sich an ein Optimum angleichen (b).

5.2.1 Erstellen von Zellen / die Zellenfabriken

Die neuen Zellen werden von einer Zellenfabrik erzeugt. Diese Zellenfabrik ist ein Zellenhaufen, welcher speziell zum Bau von anderen Zellen konstruiert wurde. Wann die neuen Zellen erstellt werden, kann nicht eindeutig definiert werden. Es gibt Zellen, die eine Herstellungspriorität haben, aber schlussendlich entscheidet die Fabrik, welchen Auftrag sie zuerst erledigt. Sobald die Zelle erstellt wurde, wird sie automatisch in Betrieb genommen, und kann nun ihre Aufgabe aufnehmen.

Die Zellenfabriken werden durch ein weiter unten beschriebenes Botschaftssystem benachrichtigt, welche Zellenart erstellt werden muss. Sind gewisse Zellen voll ausgelastet, werden Botschaften ausgeschüttet, die von den entsprechenden Fabriken verstanden werden. Die Fabrik reagiert darauf und generiert neue Zellen, welche die überlasteten Zellen unterstützen sollen. So kann das System je nach Nachfrage und Auslastung selber reagieren. Aus Sicherheitsgründen hat jede Nachfrage eine obere Grenze, die das System nicht überschreiten darf. Folgendes Beispiel soll dies verdeutlichen:

Eine Zelle ist falsch programmiert und deshalb immer voll ausgelastet. Die Fabrik erzeugt nun weitere Zellen, die die Arbeit der überlasteten Zelle versuchen zu teilen. Jede Zelle ist nun durch diesen Fehler auch voll ausgelastet. Solche Zellenarten nennen wir Wucherzellen. Ohne obere Schranke würde die Fabrik immer mehr Zellen produzieren, bis die Ressourcen des Systems aufgebraucht worden sind. Solche Zellen werden vom System speziell behandelt, wenn das System diese als Wucherzelle erkennt.

Es gibt eine weitere Art von Zellen, die egoistischen Zellen. Dies sind Zellen die Ihre Arbeit nicht teilen können oder wollen. Sie sind weit weniger schlimm als Wucherzellen. Von ihrer Gattung existieren noch inaktive Zellen, und somit generiert die Fabrik keine neuen Zellen. Einzig die Performance der Arbeit leidet darunter. Die Philosophie des System wird hier nicht eingehalten, die Arbeit zu teilen und somit schneller zum Ziel zu kommen (Siehe Arbeitsweise von Zellen).

Das Prinzip der Zellenfabriken kann vor allem für dynamische Zellen verwendet werden. Die dynamischen Zellen sind Zellen, die nicht fest zu einem Zellenhaufen gehören, sondern ähnlich wie Blutzellen zum Beispiel, im System Botschaften und Daten übertragen. Oder Zellenarten wie die Fresszellen und Polizeizellen, welche durch eine Fabrik in das System eingeschleust werden.

5.2.2 Klonen von Zellen

Zellen haben die Fähigkeit sich selber auch zu vermehren. Es gibt so genannte Mutterzellen, die sich bei Überlastung teilen können. Pro Zellensystem existiert eine Mutterzelle, die die gesamte Fähigkeit des Teilsystems besitzt. Durch das Teilen entstehen neue Zellen, die beim Erledigen der Aufgabe mithelfen. Die geklonten Zellen sind spezialisierte Zellentypen, die nur ein ganz bestimmten Teil einer Aufgabe erledigen können. Ihre Fähigkeit ist gegenüber der Mutterzelle eingeschränkt. Sie können aber bei Bedarf auch ihren spezialisierten Teilbereich wechseln, und so anderen Zellen in diesem Zellensystem aushelfen.

Das Prinzip des Klonen kann vor allem bei statischen Strukturen (Zellenhaufen) eingesetzt werden. Stellt man sich ein Baum mit verschiedenen Knoten vor, so kann der Knoten sich nun teilen und vermehren. Hier wäre eine Zellenfabrik weniger geeignet, da die Fabrik wissen müsste, zu welchem Zellenhaufen die Zellen zu erstellen sind, und wie sie dort angebunden werden müssten. Die Mutterzelle hingegen ist bereits im Haufen vorhanden und kennt die Struktur, bzw. den Ort, wo die neue Zelle ihren Platz findet.

5.2.2.1 Vermehrung in dem Zellenhaufen

Gehen wir einmal davon aus, dass die Arbeit, welche eine Zelle verrichten muss, in Form von Meldungen (Botschaften) existiert. Wir benötigen dazu einen Eingangspuffer, in welchem alle Meldungen zwischengespeichert werden, welche noch nicht von der Zelle verarbeitet wurden. Die Grösse des Puffers dient als wichtigste Grösse, um die Auslastung zu berechnen.

Der Algorithmus benötigt folgende Eingabeparameter

- Anzahl Meldungen eines Zellensystems zu einem beliebigen Zeitpunkt t: $d(t)$
- Maximale Grenze des Meldungseingangs (Puffers) N_{MAX}
- Reaktionsgrenze N_{react}

Dabei wird der Zuwachs der Meldungen und die Änderung des Zuwachs geprüft, wenn die Grösse des Eingangspuffer die Reaktionsgrenze überschreitet. Zellen werden geklont, wenn der Zuwachs positiv und die Änderung des Zuwachs stabil oder zunehmend ist.

Die Funktion $f_k(n)$ bestimmt, wann eine Zelle im Zellensystem geklont werden soll. Resultiert eine 1 wird eine neue Zelle generiert, bei 0 wird das System nicht verändert.

$$f_k(t) = \begin{cases} 1 & , \textbf{\textit{if}} \ (\ddot{d}(t) \geqslant 0 \wedge \dot{d}(t) > 0 \wedge d(t) > N_{react}) \vee d(t) > N_{MAX} \\ 0 & , \textbf{\textit{else}} \end{cases}$$

In der obigen Formel ist die Messzeit analog. Die Messzeit muss für den Computer noch diskretisiert werden. Halten wir die Messzeit variabel, so passt sie sich der Reaktionszeit des Zellenhaufens an. Für die Messzeit können eine obere und untere Schranke angegeben werden, T_{MAX} und T_{MIN}. Dies ist notwendig, um auf variable Verarbeitungszeit der Meldungen reagieren zu können. Die Messzeiten haben einen direkten Einfluss auf die Reaktionszeit des Systems. Bei langer Verarbeitungszeit der Meldungen

macht es wenig Sinn, schnelle Messungen durchzuführen. Dies würde höchstens die Performance des Systems beeinträchtigen. Im Gegensatz ist es nicht optimal, bei kurzer Verarbeitungszeit der Meldungen nur gelegentlich zu Messen. Das Wachstum würde sehr träge sein und es würde lange dauern, bis das System sich stabilisiert. Deshalb ist eine variable Messzeit sehr nützlich, um das System zeitoptimal zu steuern.

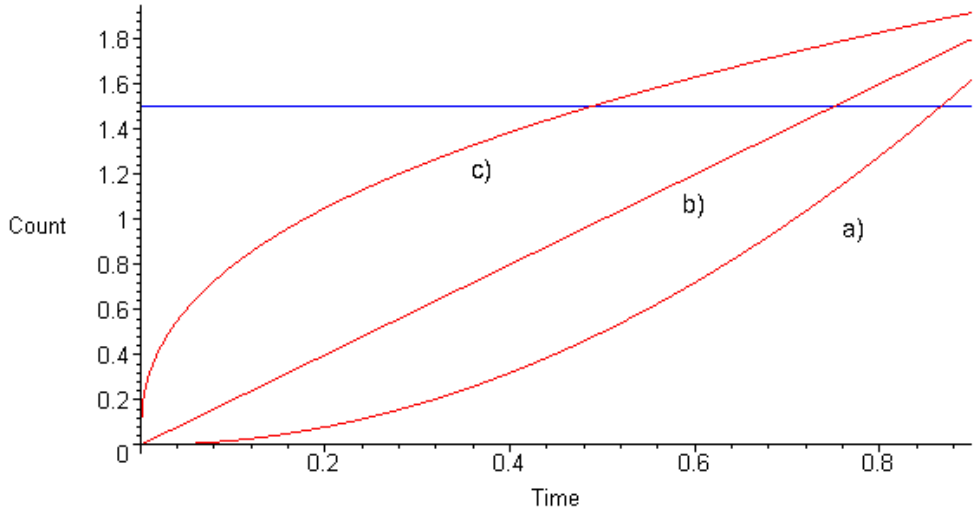


Abbildung 9: Funktionsweise des Klonalgorithmus. Ist ein Anstieg der Meldungen vorhanden, wird geprüft, ob die Veränderung positiv (a) oder gleich bleibend ist (b). Nur in diesen 2 Fällen wird geklont. Im Falle (c) wird erst geklont, wenn die Grenze überschritten wurde.

Gibt die Funktion $f_k(n)$ den Wert 1 zurück, was bedeutet, dass die Zelle geklont wird, so kann nun eine noch inaktive Zelle wieder aktiviert werden. Sind keine inaktiven Zellen mehr vorhanden, dann muss eine neue gleiche Zelle erstellt werden.

5.2.3 Abbauen von Zellen

Die Lebensdauer einer Zelle ist nicht eindeutig festlegbar. Werden Zellen über längere Zeit nicht gebraucht, werden sie zuerst inaktiv, später sterben sie ab. Dadurch kann das System auch langfristig Ressourcen freigeben, wenn die Auslastung tief ist.

Dieser Prozess schliesst den Kreislauf der Zellen. Das System wächst also, wenn die Auslastung gross ist und schrumpft, wenn die Auslastung tief ist. Diese Veränderung darf natürlich nicht schnell erfolgen, sondern entspricht einen Trend über längere Zeit.

5.2.3.1 Die Lebenskraft

Jeder Zelle wird eine Lebenskraft zugewiesen, welche beim Erstellen der Zelle das Maximum L_{MAX} enthält (zum Beispiel den Wert 100). Die Lebenskraft wird durch Verarbeiten von Meldungen um einen konstanten Wert erhöht L_{add} .

Ist keine Meldung zum Bearbeiten vorhanden, hungert die Zelle, sie liegt brach. Dabei wird die Lebenskraft um einen konstanten Wert verringert L_{sub} . Mit abnehmender Lebenskraft nimmt die Wahrscheinlichkeit zu, dass die Zelle inaktiv wird.

Die Funktion $h(t)$ liefert die Anzahl Arbeitspakete, die von einer Zelle bearbeitet wurden.

Der Algorithmus verwendet folgende Eingabeparameter:

- Lebenskraft der Zelle in Abhängigkeit der Zeit t: $g(t)$
- Erhöhung der Lebenskraft L_{add}
- Verringerung Lebenskraft L_{sub}
- Faktor für Wahrscheinlichkeit des Abbaus ω

$$g(t) = g(t-1) + \begin{cases} L_{add}, if\ h(t) > 0 \\ -L_{sub}, else \end{cases}$$

Die Funktion $f_a(t)$ gibt 1 zurück, wenn die Zelle inaktiv werden muss, sonst 0.

$$f_a(t) = \begin{cases} 1, if\ L_{MAX} \cdot e^{-g(t)/\omega} > \psi(L_{MAX}) \\ 0, else \end{cases}$$

wobei Ψ eine Random-Funktion ist die einen Wert zwischen 0 und L_{MAX} zurück liefert.

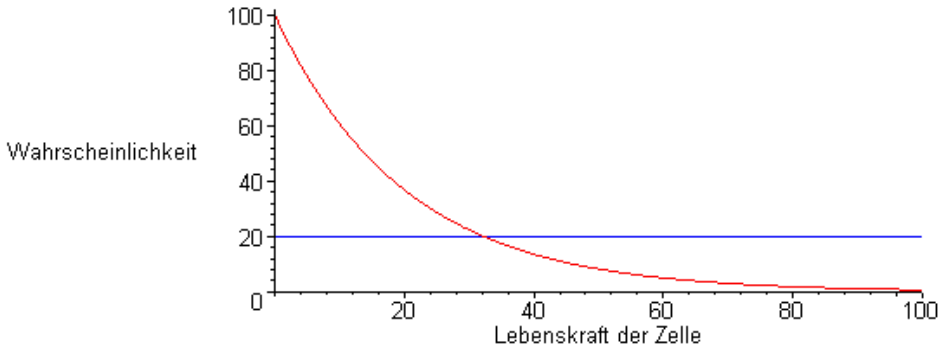


Abbildung 10: Darstellung der Wahrscheinlichkeitsfunktion in Bezug zur Lebenskraft einer Zelle

Durch eine simple Auszeit (Expire Time) werden die inaktiven Zellen dann schlussendlich zerstört.

5.2.3.2 Absturz durch Fehler

Zellen können aber auch sterben, wenn sie einen groben Fehler im System verursachen, oder wenn sie wegen eines Programmierfehlers abstürzen. Das heisst, was heute in der Computerwelt zu einem Programmabsturz führt, führt hier zum Absterben einer Zelle.

Einen anderen schwerwiegenden Fehler kann geschehen, wenn eine Zelle korrupte Daten produziert, oder durch einen Angriff von Aussen infiziert wurde. Sie überschreitet dadurch ihre Kompetenz und wird als Fremdkörper im System erkannt. Ebenso werden Zellen unbekannter Herkunft als Fremdkörper identifiziert. In diesem Falle kommen die Polizeizellen zum Einsatz. Diese Zellen sind ausschliesslich damit beschäftigt, die Sicherheit im System zu gewährleisten. Alle Zellen werden durch die Polizeizellen analysiert. Halten sie bestimmte Richtlinien nicht ein oder überschreiten ihre Kompetenzen, werden sie neutralisiert. Viren und Krankheiten werden als Erfahrungswerte gespeichert, damit sie beim Wiederauftreten schneller erkannt werden können. Die neutralisierten Zellen gelten dann ebenfalls als abgestorbene Zellen. Das Sicherheitssystem soll wie ein Immunsystem funktionieren und ist im weitesten Sinne ein integriertes Antivirus-Programm.

Die abgestorbenen Zellen werden von Fresszellen gefressen, und die Ressourcen werden dem System zurückgegeben. Wie jede Zellenart werden auch die Fresszellen je nach Bedarf von einer Fabrik erzeugt. Sterben Fresszellen ab, werden sie von ihren Artgenossen ebenfalls weggeräumt. Die Fresszellen können auch als Destruktoren bezeichnet werden. Sie bilden den Gegenpol zu den Fabriken und dem Klonen und halten so das Gleichgewicht.

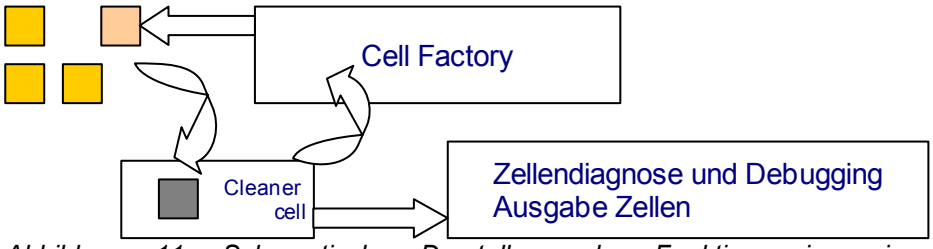


Abbildung 11: Schematische Darstellung der Funktionsweise einer Zellenfabrik

Durch ein evolutionäres Prinzip kann das Gesamtsystem gewisse Zellen verbessern, verändern oder sogar züchten. Dabei sollen unter anderem Daten beim Absterben einer Zelle generiert werden. Diese Daten stehen nach aussen zur Verfügung und können von Fresszellen gelesen und analysiert werden. Diese Daten werden dann zum Verbessern aber auch zum Debuggen des Systems verwendet.

5.3 Der Regenerierungseffekt

Der Begriff Regenerierung steht für die Fähigkeit eines Systems oder Organismus, sich selber zu reparieren, wenn irgendwo durch äussere Einwirkungen ein Schaden entstanden ist. Dabei ist es wichtig, dass der Schaden sich möglichst nur lokal auswirkt, und das System sofort wieder einen stabilen Zustand erreicht. Das Prinzip der Zellen ist hier sehr geeignet, da die Abstützung sehr breit ist. Ähnlich wie beim Internet: steigt irgendwo ein Server aus, stirbt nicht gerade das ganze Netz.

In dem CCM sieht der Regenerierungsprozess folgendermassen aus: steigt eine Zelle aus, muss diese Zelle ersetzt werden, damit das System weiterarbeiten kann. Um das Problem kurzfristig in den Griff zu bekommen und keinen grossen Leistungsverlust erleiden zu müssen, übernimmt eine benachbarte Zelle die Arbeit der defekten (zerstörten) Zelle. Das bedingt,

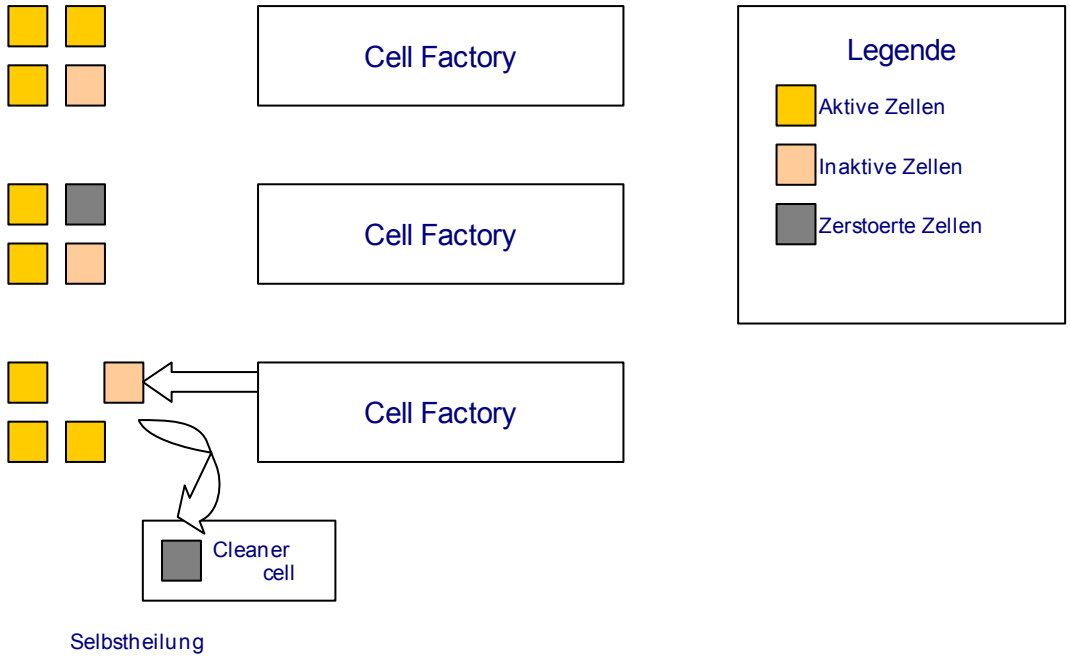


Abbildung 12: Regenerierung von Zellen. Mit Hilfe von Zellenfabriken und Fresszellen wird der Lebenszyklus von Zellen beeinflusst. Nach dem gleichen Prinzip geschieht die Regenerierung durch Klonen.

dass es im System noch inaktive Zellen hat, welche bei Bedarf sofort eingesetzt werden können. Gerade bei wichtigen Aufgaben ist es wichtig, dass die verstorbenen Zellen sofort ersetzt werden können, und nicht auf die Erzeugung neuer Zellen gewartet werden muss.

6 DNA-Der Programmcode der Zelle

Wie weiss nun die Zelle, wie sie eine Aufgabe wahrnehmen und erledigen soll? Jede Zelle besitzt ein DNA (*deoxyribonucleic acid*). Die DNA entspricht dem Programm der Zellen. Dieses Programm beschreibt die Verhaltens und Verarbeitungsalgorithmen. Jede Zelle besitzt immer die ganze DNA des Systems. Mit anderen Worten könnte eine Zelle jede Aufgabe wahrnehmen. Durch den Teilungsprozess werden jedoch den einzelnen Zellen Bereiche der DNA zugeordnet. Die Zellen kennen zwar die ganze DNA, arbeiten aber ausschliesslich im zugeordneten Bereich. Theoretisch wäre es denkbar diesen Bereich beliebig zu wechseln. Um die DNA zu interpretieren existiert in jeder Zelle ein DNA-Interpreter. Dieser Interpreter erstellt nach der DNA die Algorithmen, welche die Zelle verwendet, um ihre Aufgaben wahrzunehmen. Die DNA beschreibt auch wie diese Algorithmen mit einander agieren und kommunizieren.

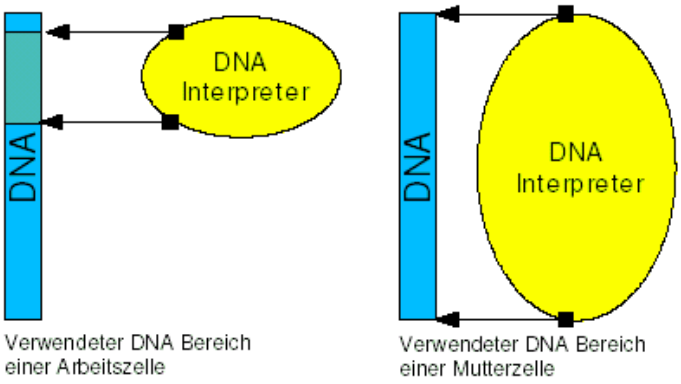


Abbildung 13: Funktionsweise eines DNA-Interpreter. Bei spezialisierten Zellen (links) wird nicht die gesamte DNA verwendet. Mutterzellen können Aufgaben von verschiedenen Zellen übernehmen (rechts), da sie grössere Ausschnitte aus der DNA interpretieren können.

Wie die DNA im Detail realisiert wird, ist noch offen. Man könnte die gesamte Information jeder Zelle mitgeben. Dieses Prinzip wird in der Natur verwendet. Der Nachteil ist, dass dies enorm viel Speicherplatz verschwendet. Die Natur komprimiert deshalb den nicht verwendeten Teil der DNA. Eine weitere Möglichkeit wäre das Halten von globalen Daten. Dadurch würde das Speicherproblem weitgehend gelöst. Es hat aber der grosse Nachteil, dass sich die DNA durch die zentrale Haltung ein gutes Angriffsziel ist. Zudem ist hier die Realisierung von Verteilten Systemen sehr aufwendig.

7 Immunsystem-Sicherheit im System

Ein Immunsystem dient dem automatischen Erkennen von Viren (schädlichen Programmen) und trojanischen Pferden (Spionage). Schlussendlich muss das Immunsystem die geeigneten Gegenmassnahmen treffen. In diesem Kapitel werden einige Verfahren vorgestellt, wie ein solches Immunsystem im Cell Computing Model realisiert werden könnte.

7.1 Erkennen von Fremdzellen

Fremdzellen sind Zellen, die nicht zum eigentlichen System gehören und somit als schädlich oder unsicher eingestuft werden. In der Biologie werden solche Zellen anhand der molekularen Struktur erkannt, sie riechen anders als die körpereigenen Zellen. Dies ist in der Informatik nicht so leicht nachzubilden. Wir können aber eine verwandtes Prinzip anwenden, den Fingerabdruck.

Mit einem Fingerabdruck (einer One-Way-Hash-Function) können wir den Programmcode einer Zelle eindeutig erkennen.

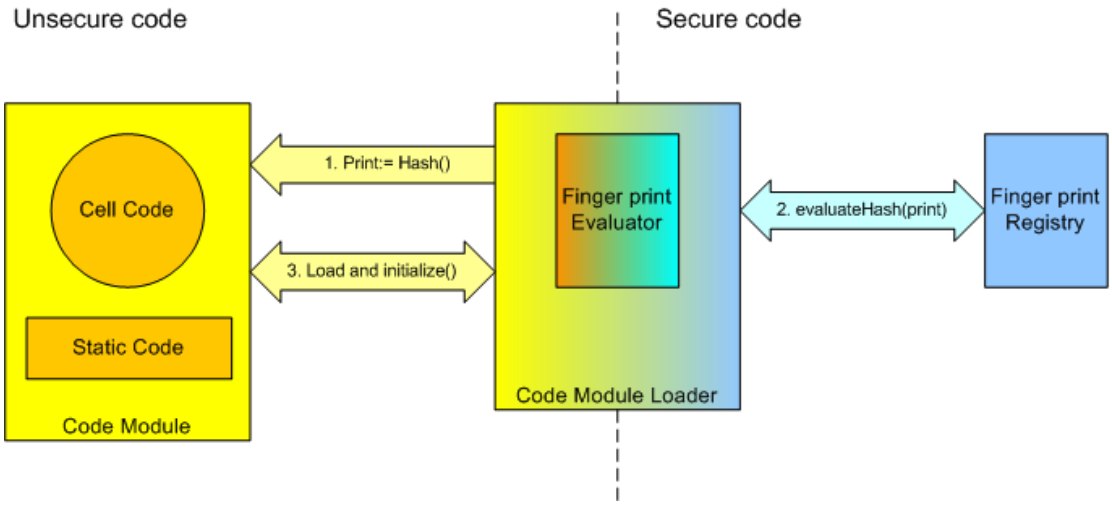


Abbildung 14: Nach dem Berechnen und Vergleichen des Finger Print kann das Code-Modul mit der Zelle geladen werden.

Dabei prüfen wir nicht nur den Programmcode der Zelle, sondern auch den zusätzlichen statischen Programmcode, da sich auch hier schädliche Programmteile verstecken könnten (Abbildung 1). Beide, der Programmcode der Zelle und der statische Programmcode, sind in einem Code-Modul zusammengefasst. In diesem Zusammenhang sprechen wir in der Informatik auch von Packages oder Shared Libraries.

Der Fingerabdruck (Finger print) wird nach dem Berechnen mit einem Eintrag einer Datenbank (Finger print registry) verglichen. Stimmt der Fingerabdruck mit dem registrierten Fingerabdruck überein, kann das Code-Modul geladen werden. Stimmt der Fingerabdruck nicht mit dem Eintrag der Datenbank überein, wird das gesamte Code-Modul als unsicher eingestuft. Das Code-Modul kann aber trotzdem geladen werden, wenn der Entwickler des Systems dies zulassen möchte.

7.2 Mobile Sicherheitszellen

Das Erkennen der Fremdkörper ist in dem Zellensystem ein wichtiger Bestandteil, um das System längerfristig am Leben zu erhalten. Um die Sicherheit möglichst breit abzustützen, wird eine dezentrale Eingreiftruppe geschaffen. Sie ermöglicht es, schnell und effizient einzugreifen, wenn ein Angriff auf das System erfolgt. Diese Eingreiftruppe besteht aus so genannten Sicherheitszellen, die andere Zellen überwachen und analysieren können. Diese Zellen können sich im System frei herum bewegen, sie sind mobil.

Beim Auftreten eines Problems können durch Trendkommunikation Sicherheitszellen angelockt werden, die versuchen das Problem zu analysieren und erste Massnahmen zu ergreifen. Eine solche Analyse kann unter anderem auch die Prüfung des Fingerabdrucks (siehe oben) beinhalten. Kann das Problem nicht ausreichend schnell oder effizient gelöst werden, können spezialisierte Sicherheitszellen angefordert werden. Das geschieht wieder durch die Trendkommunikation zu nahe liegenden Sicherheitsstellen (Zellenfabriken die Sicherheitszellen produzieren).

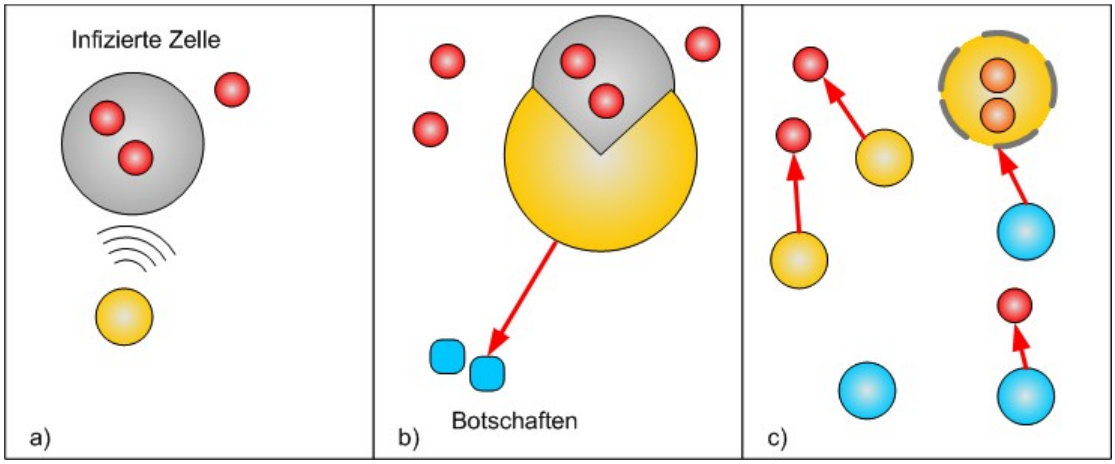


Abbildung 15: Das Immunsystem des CCM. (a) Eine Sicherheitszelle detektiert eine von Viren befallene Zelle. (b) Die Sicherheitszelle sendet Botschaften, welche andere Sicherheitszellen anlocken und versucht die befallene Zelle zu vernichten. (c) Spezialisierte Sicherheitszellen neutralisieren die Viren und die defekten Sicherheitszellen.

So genannte Gedächtniszellen, welche den Vorfall miterleben und überleben, speichern die Signatur der Virenzellen ab. Durch ihre Alarmbereitschaft, können nun Virenzellen schneller detektiert und bekämpft werden.

8 Kommunikation zwischen den Zellen

Ein sehr wichtiges Konzept ist die Kommunikation. Einerseits ist es wichtig, Zellen mit Informationen und Daten zu versorgen, die sie verarbeiten sollen (die Nahrung der Zelle), und anderseits müssen die Zelle dem System Rückmeldungen geben können. All dies geschieht mittels Botschaften (Events). Die Botschaften sind einfache Zellen oder Objekte, die Daten enthalten.

Es gibt zwei Arten von Kommunikationssysteme:

- Das Direct Communication System (DCS)
- Das Open Communication System (OCS)

8.1 Direct Communication System DCS

Das Direct Communication System DCS dient der gezielten und schnellen Kommunikation zwischen einzelnen Zellensystemen. Um ein bestimmtes Zellensystem zu erreichen muss eine Adressierung erfolgen. Der Kommunikationspartner muss bekannt sein.

Das DCS eignet sich besonders zum Steuern der Zelle oder zum Senden von Botschaften nach einem Prozessplan (Workflow).

8.1.1 Steuern von Zellen durch eine Zentrale

Die Zellensysteme werden von der Zentrale aus geleitet. Diese Zentrale hat direkten Zugang zu den Erfahrungswerten. Je nach Erfahrung und Grundintelligenz (Grundeinstellungen) des Systems, kann es nun reagieren und die Verarbeitungszellen steuern (Siehe Kommandozentrale). Rückmeldungen von Zellenfabriken und Verarbeitungszellen, sowie Daten von Sensoren können somit in die Kommandozentrale gelangen.

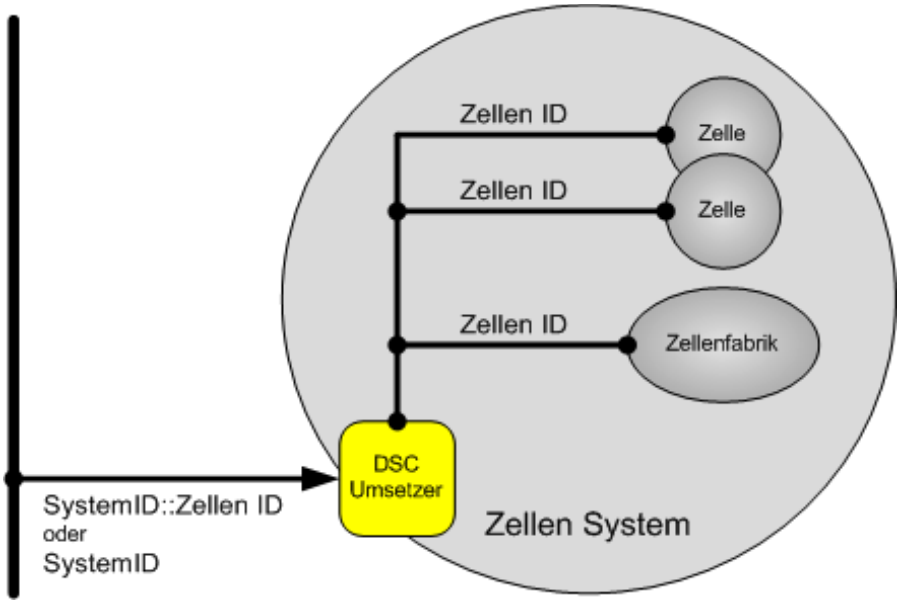


Abbildung 16: Das Direct Communication System DCS dient als direkter Draht zu Zellensystemen. Es dient der schnellen Kommunikation zwischen Zellen und Kontrollzentren

Design Hinweis: Für eine mögliche Umsetzung könnte das Soap-Protokoll von Nutzen sein. Die Adressierung der einzelnen Systeme und Zellen kann dem TCP-IP Standard angelehnt sein.

8.1.2 Workflow

Workflows finden wir im täglichen Leben. In einer Firma sind die Workflows beliebte Instrumente, um den Werdegang eines Produkts visuell darzustellen. Hier beschreibt der Workflow, wann welches Teilprodukt wo verarbeitet werden soll. Es gibt Stellen, wo eine Abteilung auf eine andere warten muss, um bestimmte Teilprodukte wieder zu eine Ganzen zusammen zu bauen. Auch können redundante Wege definiert werden, Stellvertreter die im Falle von Krankheit oder Ferien einspringen können.

Im Cell Computing Model können wir auch Workflows definieren, um dem System zu sagen, wann welches Arbeitspaket wohin gesendet werden soll. Der Workflow wird also mit dem Arbeitspaket mitgeliefert. Nähere Informationen sind im Kapitel „Arbeitsweise der Zellen“ zu finden.

8.2 Open Communication System OCS

Das Open Communication System dient im Gegensatz zum DCS als Trendkommunikation (langsame Kommunikation). Die Zellen werfen Meldungen in das Central Communication Network aus. Diese Meldungen sind nicht explizit adressiert. Jeder kann die Meldungen lesen. Wer sie interpretieren und umsetzen kann, entfernt die Meldung (verspeisen).

Ein Vorteil des OCS ist, dass verschiedene Zellen oder Zellensysteme auf Meldungen fast gleichzeitig reagieren können. Solche Meldungen werden auch fortlaufend von den Fresszellen abgebaut, wenn sie nicht gelesen werden.

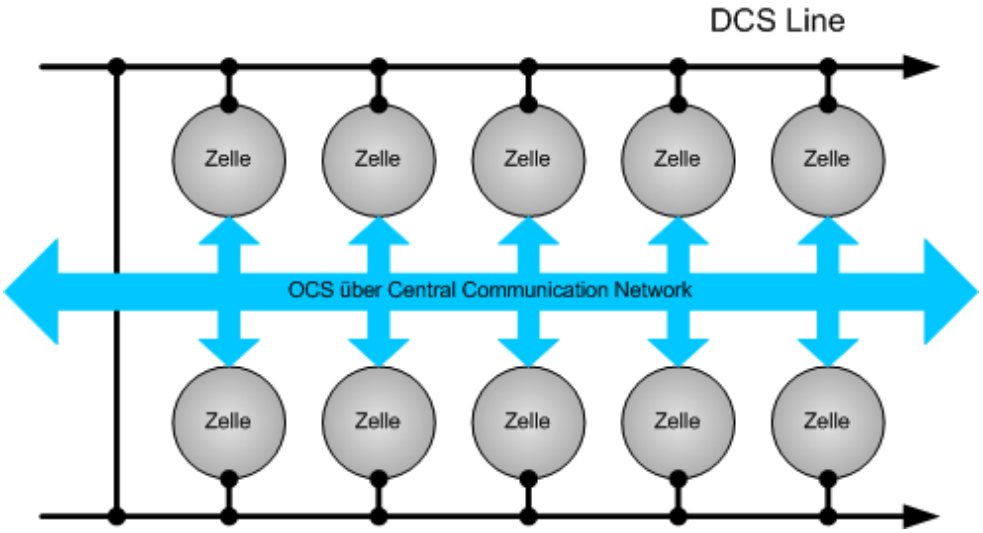


Abbildung 17: Das Open Communication System OCS dient zur Trend Kommunikation, welche längerfristige Änderungen im System steuert.

Dieses System hat aber auch einen Nachteil. Der Sender hat keinen Einfluss, wann und wo seine Meldungen gelesen werden. Es muss mehrmals dieselbe Meldung versendet werden, da unter Umständen nie alle Meldungen ihr Ziel erreichen. Ob eine Meldung verstanden und umgesetzt wurde, kann der Sender erst feststellen, wenn im System die gewünschte Reaktion erfolgt.

8.2.1 Das OCS nach dem Pool-Prinzip

Eine konkrete Art der OCS-Umsetzung ist das Pool-Prinzip. Der Pool beinhaltet alle Meldungen. Sie schwirren da sozusagen frei herum. Diese Meldungen beinhalten Daten, welche für eine bestimmte Zellenart oder einer Gruppe von Zellen definiert sind.

Damit nun jeder Zellenhaufen an diese Meldungen gelangen kann, existiert einen globalen Zugriffskanal auf den Pool. Durch diesen Kanal kann jede Zelle Meldungen absetzen aber auch Meldungen anfordern und lesen. Eine Zelle wählt eine beliebige Meldungen im Pool aus. Kann sie die Meldung interpretieren, reagiert sie entsprechend darauf und hat somit diese Meldung verbraucht. Andere Zellen können auf diese Meldung nicht mehr reagieren. Kann eine Zelle die Botschaft nicht interpretieren, gelangt sie unverändert zurück in den Pool. Wie jede Zelle hat auch die Meldung eine Lebensdauer. Wird sie nicht gelesen oder niemand kann sie interpretieren, wird sie nach einer gewissen Zeit von Fresszellen gefressen, und die Ressourcen werden freigegeben.

Der Code der Meldung legt fest, welche Zellenart sie verbrauchen darf. Aber welche der Zellen gerade damit beschäftigt ist die Meldung wirklich zu verarbeiten, lässt sich nicht vorhersagen.

Ein Nachteil des Pools ist, dass es sich hier um eine zentrale und somit verletzliche Komponente handelt. Zentrale Objekte sind immer ein gutes Angriffsziel von Viren und Hacker.

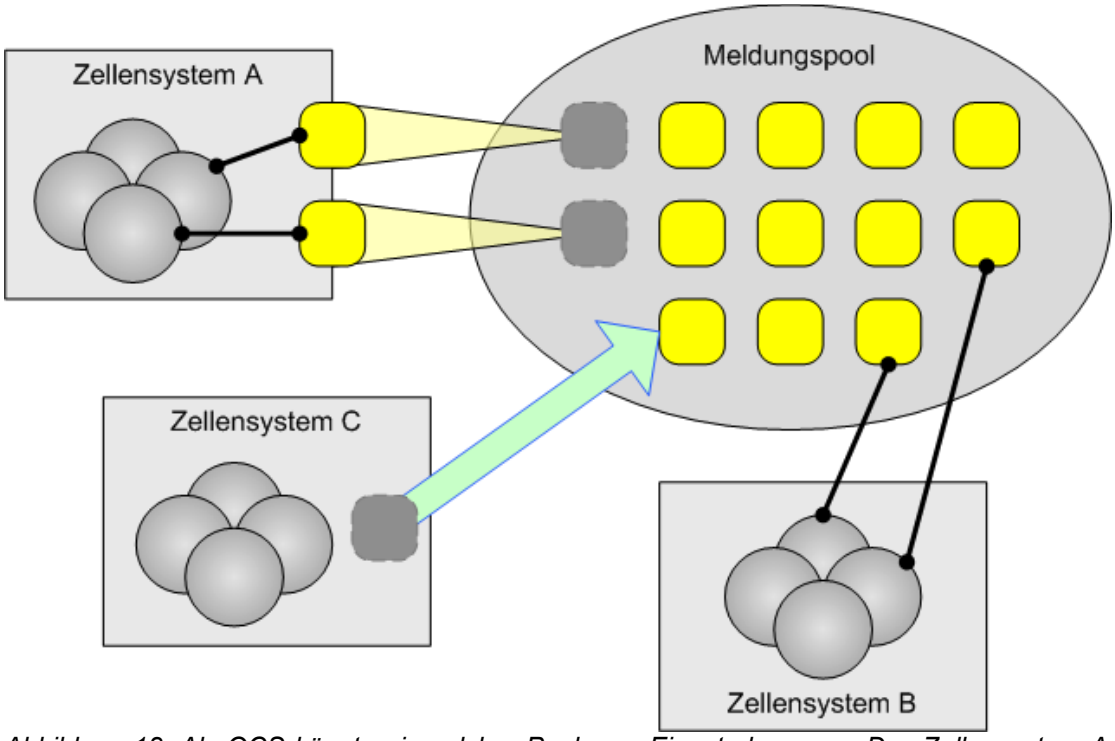


Abbildung 18: Als OCS könnte ein solcher Pool zum Einsatz kommen. Das Zellensystem A entfernt Meldungen zum Verarbeiten. Das Zellensystem B prüft, ob es die Meldung gebrauchen kann und das Zellensystem C sendet Meldungen an den Pool.

8.3 Beispiel: Hacker Angriff

Das folgendes Beispiel soll die Funktionsweise des OCS verdeutlichen:

Durch einen Angriff von aussen (durch Hacker oder Viren) wurden einige Zellen zerstört, auch drangen Virenzellen in das System ein, um weiteren Schaden zu verursachen. Die Sicherheitszellen haben den Angriff erkannt und sind nun beschäftigt, die Virenzellen zu eliminieren. Durch Botschaften an die Zellenfabrik, welche die Sicherheitszellen herstellt, wird bekannt gegeben, dass mehr Sicherheitszellen zu produzieren sind. Die Fabrik reagiert auf diese Botschaften und produziert nun vermehrt Sicherheitszellen. Durch den Angriff mangelt es im System an Arbeitszellen. Das System reagiert mit der Produktion dieser Zellenart auf die erhöhte Nachfrage. Während die zerstörten Zellen nun von den Fresszellen entfernt werden, übernehmen die umliegenden und neu produzierten Zellen die Produktion.

9 Arbeitsweise der Zellen

Wie arbeiten nun diese Zellensysteme zusammen, um etwas brauchbares herstellen zu können? Um dieser Frage auf den Grund zu gehen müssen wir die zu verrichtende Arbeit etwas genauer anschauen. Jeder Auftrag, jede Arbeit, jedes Problem kann in kleinere Einheiten aufgeteilt werden. Diesem Prinzip folgt jede Programmierung. Der Unterschied besteht aber in der Art und Weise der Aufteilung. Das CCM verlangt ein paralleles Denken. Mit anderen Worten, das Aufteilen der Arbeit geschieht nach dem Grundsatz: welche Teilbereiche können parallel abgearbeitet werden. Diese Teilbereiche müssen nicht einmal unabhängig von einander lösbar sein. Zwischen den parallelen Abläufen kann eine Kommunikation stattfinden, die als Synchronisation gebraucht werden kann.

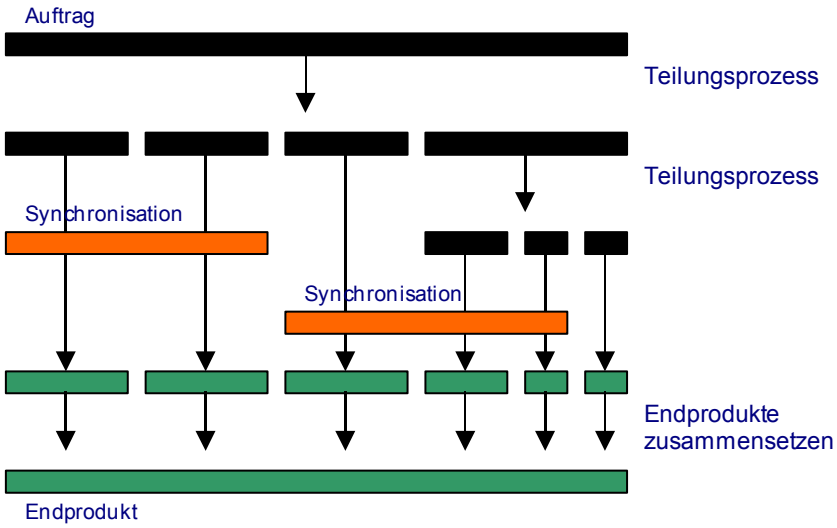


Abbildung 19: Aufteilen von Arbeitspaketen, paralleles Verarbeiten der kleinen Pakete und zusammenfügen der Resultate.

Der grosse Vorteil des parallelen Verarbeiten der Aufträge ist die enorme Zeitersparnis gegenüber dem linearen Vorgehen. Durch den Einsatz mehrerer Zellen ist der Auftrag ziemlich breit abgestützt. Das Aussetzen einer Zelle bedingt deshalb nicht einen kompletten Neubeginn, um den Auftrag zu erledigen. Eine neue Zelle führt die von der defekten Zelle hinterlassenen Arbeit zu Ende.

9.1 Aufteilen der Arbeit

Jeder Auftrag muss eine klar definierte Struktur aufweisen. Diese wird verwendet, um diesen Auftrag ordnungsgemäss aufteilen zu können. Die Aufteilung ist der erste Arbeitsschritt und wird von einer Zelle im System vorgenommen.

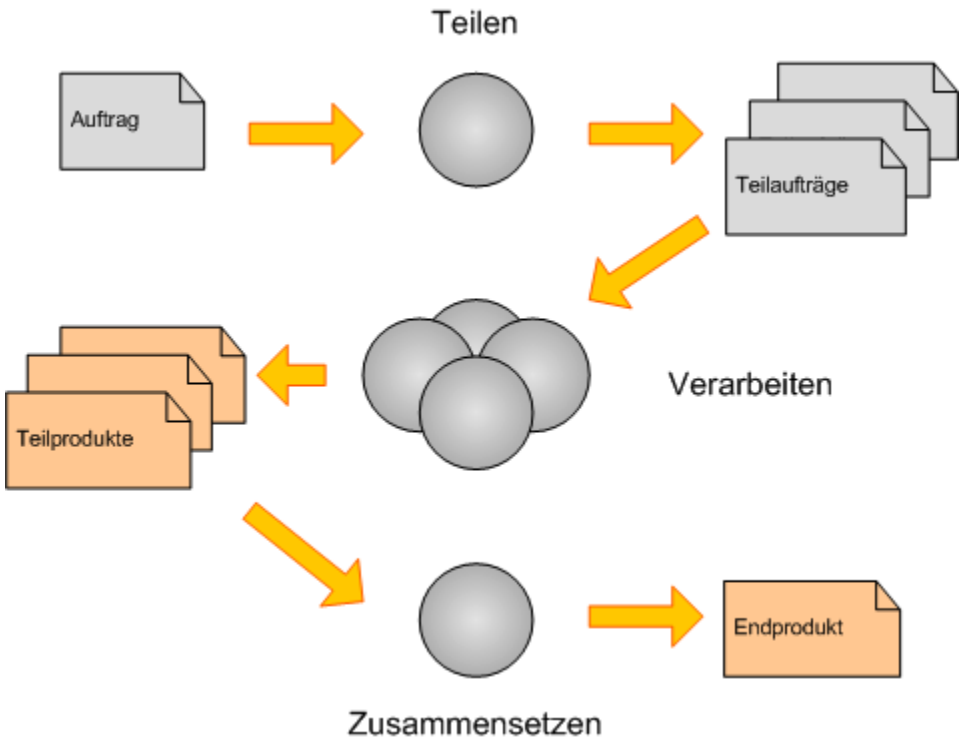


Abbildung 20: Das Aufteilen und Zusammenfügen der Arbeitspakete wird durch das Zellensystem selber vorgenommen.

Jede Zelle kann ihren Auftrag teilen, um so die Belastung herabzusetzen, indem sie Bearbeitung der Teilaufträge delegiert. Andere Zellen, sofern vorhanden, übernehmen dann diese Teilaufträge. Aus jeder Verarbeitung resultiert ein Produkt. Die Teilprodukte werden durch die Zelle, welche den Auftrag geteilt hat, wieder zusammengesetzt. Am Ende steht dann das Endprodukt. Es gibt natürlich beliebig viele Zwischenkombinationen: Ein Produkt kann zugleich ein Auftrag sein, welcher wieder von Zellen geteilt und verarbeitet werden kann.

Der Ablauf jedes Verarbeitungsprozesses muss genau geplant und nach diesem Schema umgesetzt werden. Dies kann mit Hilfe von Workflows und Petrinetzen erfolgen.

9.2 Beschreiben eines Workflows

Für jedes Arbeitspaket existiert eine genaue Verarbeitungsbeschreibung, der Workflow. Ein Workflow definiert im Cell Computing Model folgende Parameter:

- wo stehen wir in dem Verarbeitungsprozess
- müssen wir auf andere Teilprodukte warten
- an welche Zellen oder Zellensysteme muss das verarbeitete Produkt weitergesendet werden.

Um ein Workflow optimal und allgemein gültig zu beschreiben, verwenden wir Petrinetze. Petrinetze haben den grossen Vorteil, dass sie mathematisch analysiert werden können. Somit kann festgestellt werden, ob das Netz verklemmungsfrei und sicher ist.

Die Abbildung 19 zeigt ein Zellensystem A welches 4 Arbeitspakete zur Auslieferung besitzt. Die Transition (Quadrat) ist dann aktiviert, wenn die Bedingung erfüllt ist, dass heisst es müssen Vor- und Nachbereich der Transition erfüllt sein.

Eine Bedingung kann folgendes Enthalten:

- Hat es Arbeitspakete vom Typ X im Vorbereich (Zelle A). (Nach Definition vom Petrinetz)
- Hat es genügend Kapazität im Nachbereich. (Nach Definition vom Petrinetz)
- Ist das Zellensystem B erreichbar.

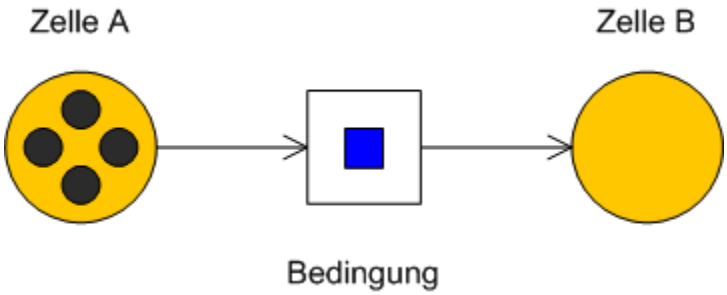


Abbildung 21: Darstellung eines Verarbeitungsprozesses mit 2 Zellensystemen

Ist die Bedingung wahr, wird das Arbeitspaket von der Zelle A an die Zelle B gesendet (Aktion).

9.2.1 Definieren von Bedingungen

Die obige Definition hat gewisse Nachteile. Wir müssen den Vorbereich kontaktieren und prüfen. Durch folgende Definition kann das Prüfen vereinfacht werden und den Ablauf schneller erfolgen:

- Fertig geleistete Arbeit wird von der Zelle A weiter geschickt. Dabei wird nur die Erreichbarkeit der Zelle B geprüft.
- Kapazität beim Zielsystem ist unendlich gross.

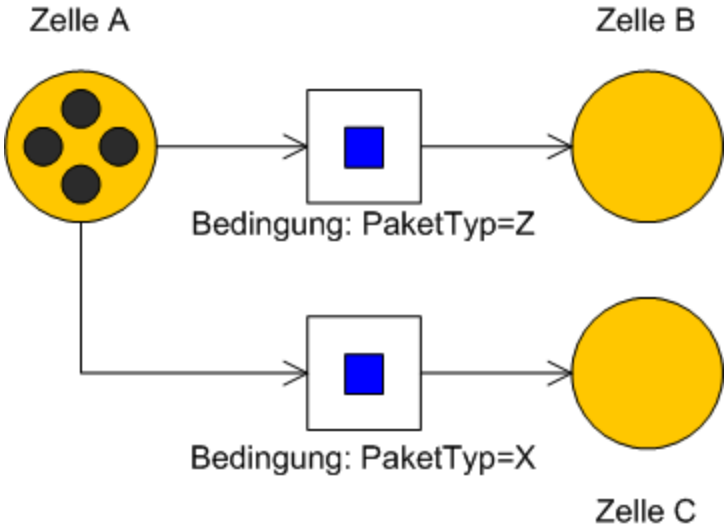


Abbildung 22: Es werden gleichzeitig die Pakete vom Typ X und Z an die entsprechenden Zellensysteme gesendet.

Zusätzlich können wir nun definieren, welche Aufträge (Tokens) eine Transition aktivieren. Dadurch können wir von einer Zelle aus definieren, dass unterschiedliche Arbeitspakete an verschiedene Zellen gesendet werden (Abbildung 21).

9.2.2 Synchronisieren

Ein wichtiger Bestandteil der Petrinetze ist das Synchronisieren und Parallelisieren von Abläufen. Das Synchronisieren wird gebraucht um Teilprodukte zusammenzusetzen.

Die Zellen A1 und A2 (Abbildung 20) senden ihre Arbeitspakete an die Zelle B, welche diese Pakete zu einem Produkt zusammenfügt.

Die gesamte Koordination beim Synchronisieren ist nicht trivial lösbar. Hier ist ein möglicher Ansatz gegeben.

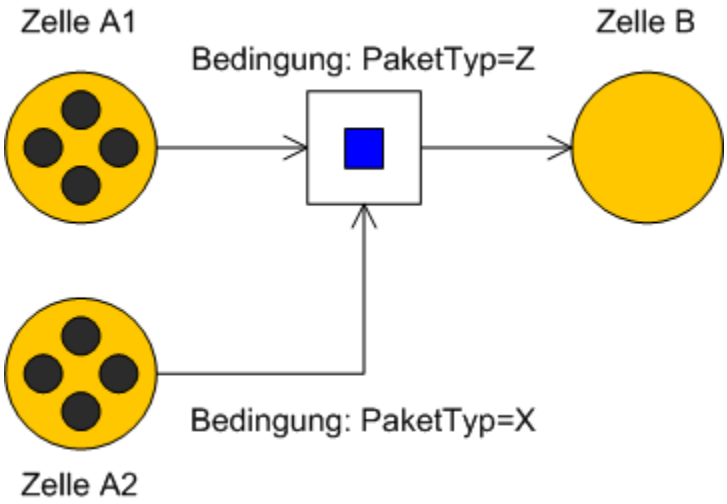


Abbildung 23: Synchronisieren der Zelle A1 und A2.

9.2.2.1 Das Synchronisationsverfahren

Da ein Netzwerk nicht als sicher eingestuft werden kann, muss eine Fehlerbehandlung eingebaut werden. Zu den obigen Punkten können verscheiden Strategien verwendet werden. Nehmen wir an die Zelle A1 ist bereit zur Auslieferung des Arbeitspakets Z.

1. Die Zelle A1 informiert die Zelle A2 mit einem Request R1, dass sie zur Auslieferung bereit ist.

   a) Die Zelle A2 kann nicht erreicht werden. Nach x-Versuchen abbrechen und durch Back Tracking den Fehler an die vorhergehenden Zellen zurückgegeben, oder den Fehler an eine Fehlerbehandlungsroutine weiterleiten.

2. Sobald die Zelle A2 bereit ist, wird ebenfalls einen Request R2 an die Zelle A1 gesendet.

   a) Die Zelle A1 ist nicht erreichbar. Gleiche Fehlerbehandlung wie bei Punkt 1.

   b) Die Zelle A1 ist erreichbar, das Arbeitspaket wurde aber verworfen, da die Zeit abgelaufen ist (expired). Die Zelle A2 muss nun ihr Arbeitspaket auch verwerfen, da es nutzlos geworden ist.

3. Steht bei einer Zelle einen Request an, und die Zelle hat selber einen Request gesendet, wird eine Answer gesendet und auf die Answer der anderen Zelle gewartet.

   a) Request wurde gesendet, es kommt aber kein Answer auf diesen Request. Nach x-Versuchen oder nach einer definierten Zeit wird abgebrochen und die Fehlerbehandlung eingeleitet.

   b) Beim Senden der Answer ist die andere Zelle nicht mehr erreichbar. Es wird abgebrochen und die Fehlerbehandlung eingeleitet.

4. Beim Eintreffen der Answers werden die Arbeitspakete gesendet.

   a) Das Zielsystem ist nicht erreichbar. Fehlerbehandlung wird eingeleitet.

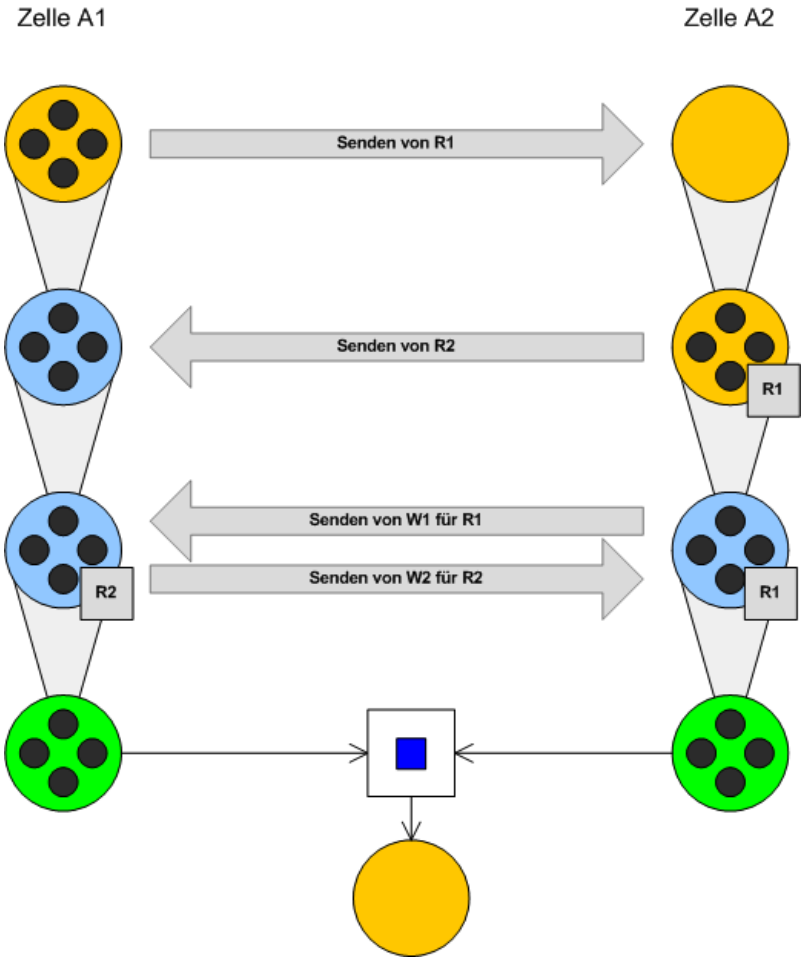


Abbildung 24: Ablauf der Synchronisation in einem Workflow. Bis zum Synchronisieren müssen die Zellen A1 und A2 untereinander ihre Bereitschaft bekannt geben.

9.2.3 Parallelisieren von Abläufen

Parallelisieren heisst, ein Arbeitspaket an zwei Zellensysteme versenden. Dabei wird das Arbeitspaket dupliziert.

Dieser Ablauf ist wesentlich einfacher. Es müssen lediglich der Vorbereich und der Nachbereich geprüft werden.

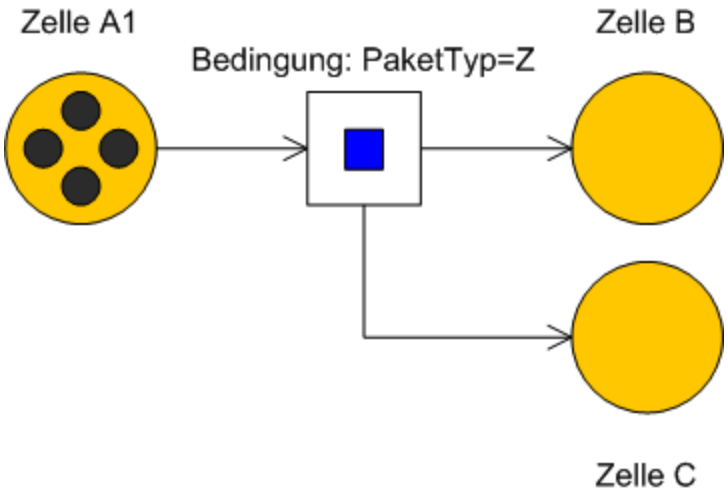


Abbildung 25: Parallelisieren von Abläufen

9.2.4 Definieren von Workflows

Durch das Verwenden von Workflows, können wir im gleichen Netz verschiedene Aufträge verarbeiten, ohne dass wir das System neu konfigurieren müssen. Der Workflow wird mit jedem Arbeitspaket mit gegeben. Somit wissen die Zellen beim Erhalten des Auftrags, wie er bearbeitet werden muss.

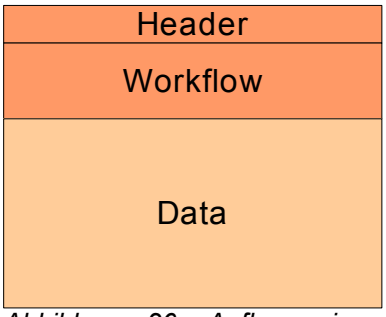


Abbildung 26: Aufbau eines Arbeitspaketes

9.3 Beispiel Versorgung mit Aufträgen

Wie gelangen Zellen an ihre Arbeit? Hier kann das DCS gebraucht werden. Jeder Auftrag hat eine Zieladresse. Diese Zieladresse definiert ein bestimmtes Zellensystem (Zellengruppe) für welche die Arbeit definiert ist. Das Zellensystem nimmt diesen Auftrag aus einem Kanal des DCS entgegen und nimmt eine gewisse Aufteilung der Arbeit vor (siehe oben). Die Teilaufgaben und Teilprodukte werden in einem internen Pool gespeichert. Jede Arbeitszelle hat einen speziellen Kanal auf diesen Pool, durch den sie die Aufträge entgegen nehmen kann. Jede Arbeitszelle prüften nun, ob ein Auftrag ansteht und nimmt diesen entgegen. Sobald der Auftrag erledigt ist, wird dieser wieder in den Datenpool des Zellensystems abgelegt. Dort wird der Auftrag entweder mit anderen Aufträgen wieder vereint oder an andere Zellen weitergeleitet. Dies geschieht nach dem Workflow. Er definiert wer, was und wie verarbeitet.

10 Kommandozentrale des Systems

Die Kommandozentrale ist die intelligente Komponente des CCM. Sie entspricht sozusagen dem Gehirn und übernimmt ähnliche Aufgaben, welche in diesem Abschnitt näher erklärt werden. Die Kommandozentrale ist noch am wenigsten beschrieben, ist aber eines der wichtigsten Komponenten des CCM. Durch sie wird das CCM erst zum System, welches autonome Fähigkeiten und künstliche Intelligenz besitzt.

10.1 Steuern der Zellen

Eine der Aufgaben ist das Steuern der einzelnen Zellensysteme. Wie bereits beschrieben, geschieht die Steuerung entweder durch direkte Kommunikation (Siehe DCS) oder durch die Trendkommunikation (Siehe OCS). Während die direkte Kommunikation dem Nervensystem bei biologischen Systemen entspricht, wird mit der Trendkommunikation eine Art Steuerung mit Hormonen realisiert.

Die Kommandozentrale empfängt Signale aus der Umgebung des Systems, wie Eingaben eines Benutzers oder Bilder von Kameras usw. Diese Signale werden von den Sensorzellen in Meldungen (Datenpakete) umgewandelt. Aus diesen Meldungen erfolgt der Start der Verarbeitung. Diese Signale werden meistens über die direkte Kommunikation erfolgen. Zuerst muss zu jeder Meldung oder Meldungsart eine geeignete Verarbeitungsbeschreibung (Workflow) hinzugefügt werden. Diese Beschreibung dient dem Verarbeiten, Aufteilen, Synchronisieren und dem Zusammenfügen mit anderen Daten. Die Verarbeitungsbeschreibung ist entweder fix codiert oder durch Erfahrung des Systems entstanden.

Über die Trendkommunikation können die Zellensysteme nachhaltig beeinflusst werden. Zum Beispiel eine Stresssituation könnte ein Trend auslösen, welcher Zellenfabriken veranlasst mehr Zellen zu produzieren. Das Feststellen, ob eine solche Situation vorliegt kann einerseits lokal erfolgen (in einem Zellensystem) oder durch globale Verhaltensmuster des Gesamtsystems. Während ersteres auch lokal gelöst werden kann, muss das Zweite durch die Kommandozentrale gelöst werden.

10.2 System Erfahrung

Eine weitere wichtige Aufgabe der Kommandozentrale ist, gewisse Entscheidungen selber fällen zu können und je nach Erfahrung selbständig Einfluss auf die jeweiligen betroffenen Zellensysteme zu nehmen. Die Kommandozentrale hat direkten Zugriff auf eine Erfahrungsdatenbank. Hier können die Erfahrungen miteinander verknüpft und gespeichert werden. Diese Erfahrungsdatenbank könnte zum Beispiel mit einem neuronalen Netzwerk implementiert werden, in dem jedes Neuron auch eine Zelle ist.

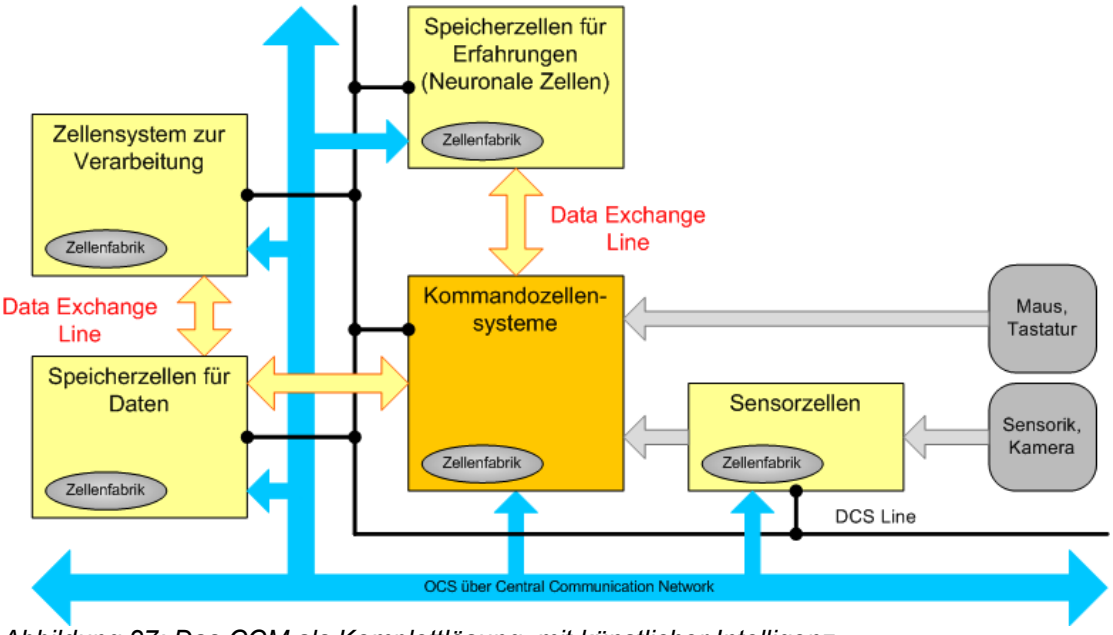


Abbildung 27: Das CCM als Komplettlösung, mit künstlicher Intelligenz.

10.3 Evolutionäre Weiterentwicklung

Die Fähigkeit gewisse Zellen zu züchten und weiter zu entwickeln wird ebenfalls durch Bereiche in der Kommandozentrale gesteuert. Wie weit dieses Prinzip eingesetzt werden kann, um die DNA der jeweiligen Zellen zu verändern, bzw. gewisse Mutationen vorzunehmen, muss noch im Detail abgeklärt werden.

11 Die Datenablage

11.1 Lokale Zellendaten

Die lokalen Zellendaten dienen der Zelle als Arbeitsdaten. Sie werden nur von einer bestimmten Zellenart verwendet und können von anderen Zellenarten nicht ohne weiteres verwendet werden. Es handelt sich hier nur um temporäre Daten, welche von einem Zellensystem verarbeitet werden. Nicht jede einzelne Zelle hat einen lokalen Speicher. Der lokale Speicher wird vom Zellensystem verwaltet. Das hat den Vorteil, dass die Daten von verschiedenen Zellen gleichzeitig verarbeitet werden können. Kann eine Zelle nun die Arbeit nicht zu Ende führen, übernimmt eine inaktive Zelle die verbleibende Arbeit. Dieses Data Sharing verlangt aber eine klare Absprache zwischen den arbeitenden Zellen.

11.2 Die Speicherzellen

Im Gegensatz zu den lokalen Daten können Daten auch global in den Speicherzellen abgespeichert werden. Die Speicherzellen stellen eine globale Datenbank dar, in welcher jeder mit Zugriffsrecht zugreifen und Daten verarbeiten kann.

Es gibt zwei Hauptgruppen von Speicherzellen:

- Speicherzellen für Verarbeitungsdaten
- Speicherzellen für die Erinnerungswerte

Die Erinnerungswerte werden von der Kommandozentrale verwaltet. Sie entstehen durch die Lernfähigkeit und den Erfahrungen des Systems. Es entsteht eine netzartige Verknüpfung zwischen diesen Daten, eine Möglichkeit gewisse Daten mit anderen Daten in Verbindung zu bringen.

Die Speicherzellen der Verarbeitungsdaten könnten mit einer relationalen Datenbank implementiert werden.

11.3 Beispiel einer Datenverwaltung

Nehmen wir an, eine Kamera generiert Bilder. Diese Bilder werden nun von den Sensorzellen zuerst so umgewandelt, dass das System sie interpretieren kann

Real World -> Hardware -> Sensorzellen (Treiber) -> Bildformat (jpg, gif etc.)

Nun sollen diese Bilder von gewissen Zellen weiterverarbeitet werden. Damit dies geschehen kann, werden sie von den Sensoren über die Kommandozentrale in Speicherzellen abgelegt. Über die Kommandozentrale werden nun die entsprechenden Zellen animiert, diese Informationen zu verarbeiten. Die Bilder werden zum Teil kopiert und als lokale Daten in der jeweiligen Zellengruppe abgespeichert. Nach der Verarbeitung der Bilder werden diese wieder in den Speicherzellen abgelegt, und stehen nun anderen Zellensystemen zur Verfügung, die sie je nach dem weiterverarbeiten oder ausgeben.

11.4 Einige Zellenarten

Arbeitszellen:	Die Arbeitszellen dienen der Datenverarbeitung des Systems. Sie entsprechen der logischen Einheit des Systems.
Sensorzellen:	Diese Zellen entsprechen der Dateneingabe eines herkömmlichen Systems.
Ausgabezellen:	Diese Zellen dienen als Schnittstelle zwischen dem Benutzer und dem System. Sie geben Aufschluss, was das System verarbeitet hat (Ausgabe).
Fabrikzellen:	Hier werden je nach Bedarf verschiedene Zellenarten produziert. Die Fabrikzellen sind lebenswichtig für das System.
Fresszellen:	Diese Zellen räumen das System auf und entfernen verwaiste und zerstörte Zellen
Sicherheitszellen:	Zellen dieser Art sind für die Sicherheit zuständig und zerstören fremde und gefährliche Zellen.

Versorgung:	Diese Zellen transportieren Meldungen durchs System (langsame Kommunikation oder Trendkommunikation)
Speicherzellen:	Speicherzellen verwalten den Datenspeicher des Systems. Sie sind also die Schnittstelle zwischen dem System und der Datenablage.
Meldungszellen:	Die Meldungszellen beinhalten die Aufträge mit der Auftragsbeschreibung (Code). Es sind sehr schlanke Zellen, die für den Transport zuständig sind.

Diplomarbeit

Zeus-Framework

12 Einleitung zum Framework

Das Zeus-Framework ist eine Implementation des Cell Computing Model (CCM). Das CCM ist aber nur eines von vielen Konzepten, die in diesem Framework umgesetzt wurden. Dieses Dokument beschreibt, was mit dem Zeus-Framework Applikationen realisiert werden kann, und wie die einzelnen Konzepte umgesetzt wurden.

Dieses Dokument beinhaltet:

- Die Spezifikation des Module Object Model (MOM) Version 2.0.
- Die Beschreibung des Klassen-Frameworks, dazu die Umsetzung des Cell Computing Model, welches in [CCM1] spezifiziert wurde.
- Die Bedienanleitung der Zeus-Framework Applikation

12.1 Das System

Das Zeus-Framework besitzt zwei Haupteigenschaften:

- modulare Struktur mit Softwarekomponenten in einem abgeschlossenen System (statische Eigenschaft)
- dynamische Kommunikation mit anderen gleichartigen und fremden Systemen. Zudem autonomes Anpassen an neue Situationen. (dynamische Eigenschaft)

Diese Eigenschaften ermöglichen ein breites und umfassendes Einsatzgebiet des Frameworks. Weitere Eigenschaften, wie

- portabel auf verschiedene Systeme (Linux und Windows bereits getestet)
- schnell und effizient (In C++ geschrieben)
- keine Lizenzgebühren (Open Source)

zeichnen das Framework ebenfalls aus.

Es setzt moderne Konzepte wie eXtensible Markup Language (XML) [XML01], Remote Method Invocation (RMI) und Objektserialisierung um und bietet eine einfache objektorientierte API (Application Programming Interface).

12.2 Einsatzgebiet

Das Zeus-Framework kann auf sehr verschiedene Arten eingesetzt werden:

- Durch seine modulare Struktur eignet es sich gut für grosse Projekte. Mehrere Entwickler können gleichzeitig an einem Produkt arbeiten. Integration von Softwarekomponenten ist sehr einfach möglich.
- Einsatz für hochdynamische, verteilte Anwendungen, bei denen künstliche Intelligenz zum Einsatz kommt.
- Es eignet sich für parallele Verarbeitung von Daten (Clustering).
- Es kann zur Entwicklung von Anwendungen dienen, bei denen Java oder C# aus Grund der schlechteren Effizienz nicht eingesetzt werden kann (Bsp. bei hardwarenaher Software)
- Bedien- und Kommunikationssoftware für Maschinensteuerungen

12.2.1 Clustering

Durch Clustering können mehrere PCs zu einem Grosssystem zusammengeschaltet werden. Diese PCs verarbeiten Daten parallel und lösen ein Problem gemeinsam. Die Arbeitslast ist je nach Problem verschieden und die PCs reagieren selbständig auf die Änderung der Auslastung. Dies ist ein typischer Einsatz des Cell Computing Model.

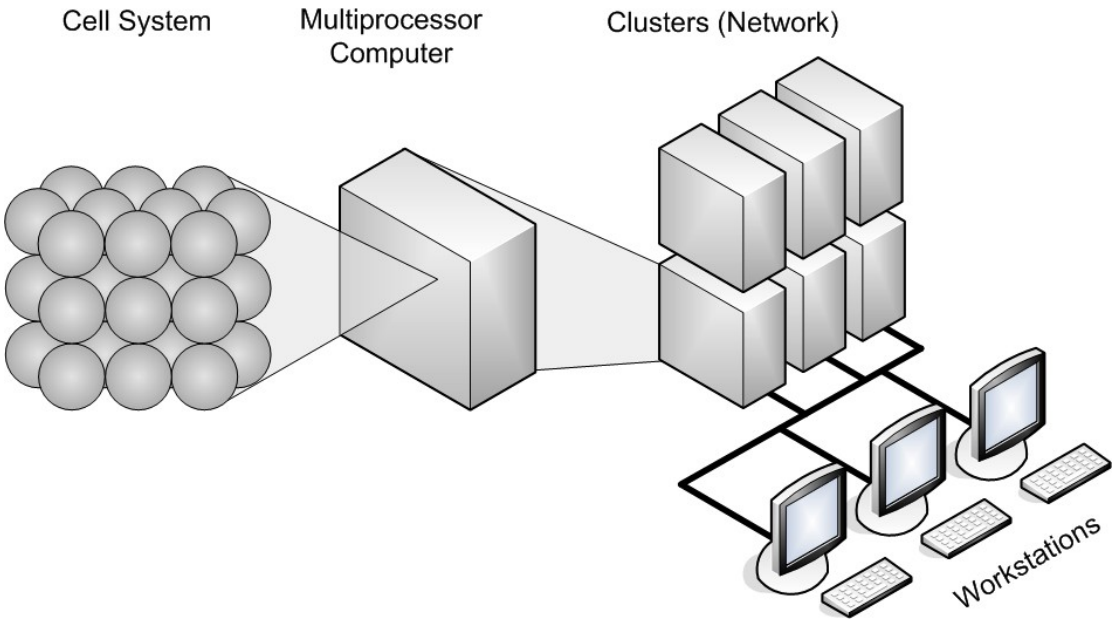


Abbildung 28: Clustering mit normalen PCs oder Workstations zu einem Grosssystem

Die Zellensysteme werden auf den PCs installiert, welche untereinander mit einem Netzwerk verbunden sind. Durch eine Registrierungsstelle (Namensdienst), welche auf einem der PCs läuft, können sich die Zellensysteme gegenseitig finden und somit miteinander kommunizieren. Die PCs, welche gemeinsam eine Registrierungsstelle benötigen, bilden einen so genannten Bearbeitungscluster.

Mehrere Bearbeitungscluster können nun über die Registrierungsstellen miteinander gekoppelt werden. Die Registrierungsstellen tauschen Informationen untereinander aus, welche die einzelnen Zellensysteme betreffen. Zellensysteme von verschiedenen Bearbeitungscluster können so miteinander kommunizieren.

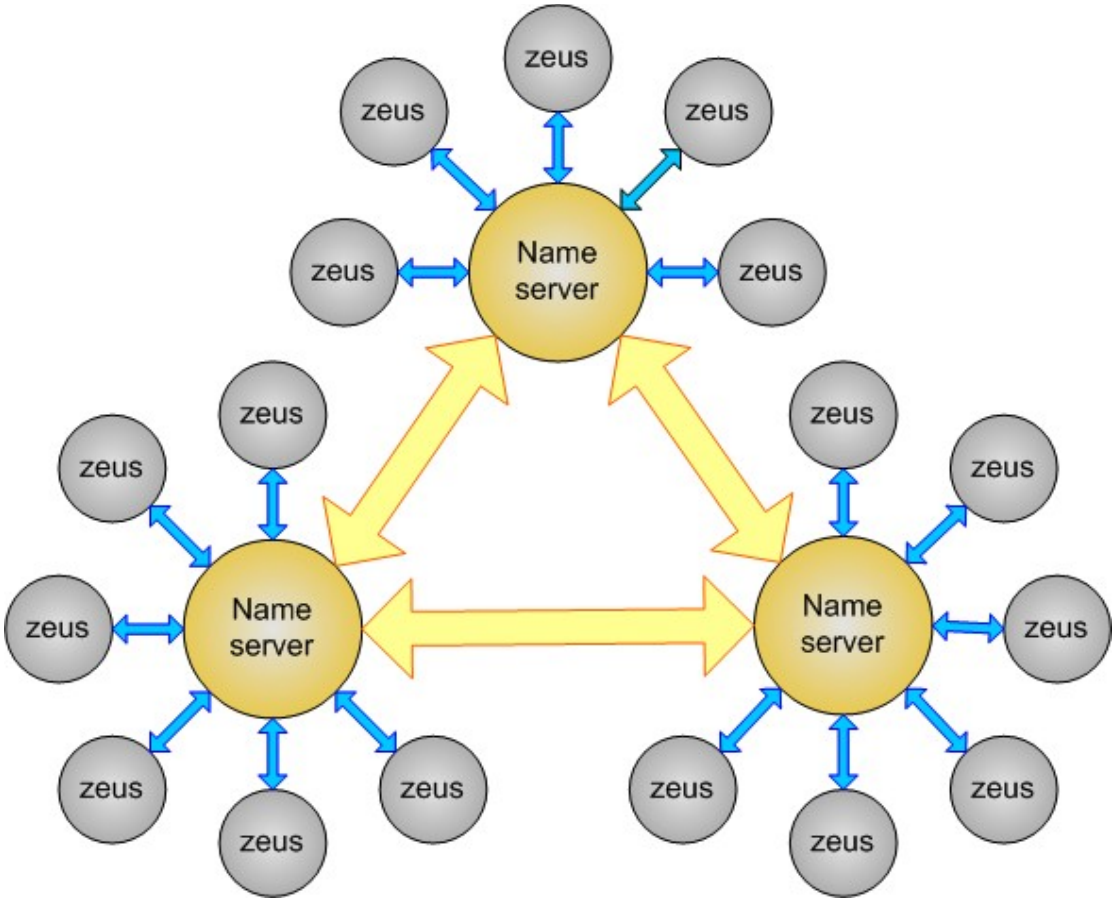


Abbildung 29: Mehrere Bearbeitungscluster werden durch die Registrierungsstellen miteinander gekoppelt. Dadurch können Systeme verschiedenster Grösse realisiert werden.

12.2.2 Bediensoftware für Maschinen

Bei der Bediensoftware für Maschinen spricht man etwa auch von Human Machine Interface (HMI). Das HMI hat die Aufgabe Maschinendaten möglichst schnell zu visualisieren, Benutzereingaben an die Maschine weiterzuleiten und Aktionen auf der Maschine auszuführen.

Das Hauptproblem ist fast immer die Zeit. Je nach Datenmenge und Art der Maschinensteuerung ist die Kommunikationsschnittstelle ein Engpass. Aber auch die Aufbereitung und das Rechnen der Daten kann sehr viel Zeit in Anspruch nehmen. Deshalb muss die Übertragung der Daten von der

Maschine zur Bedienoberfläche und zurück möglichst schnell und optimal laufen. Das Zeus-Framework wird hier einerseits bei der Darstellung und Verarbeitung der Daten und anderseits bei der Kommunikation verwendet.

Es ist vorstellbar, dass in Zukunft nicht mehr eine zentrale Maschinensteuerung mit PLC (Programmable Logic Controller) und CNC (Computer Numeric Control) existiert, sondern dass sich mehrere Prozessoren in einem Maschinenraum befinden und die Daten möglichst nahe bei der Messstelle auswerten. Hier könnte das Zeus-Framework ebenfalls zum Einsatz kommen. Durch sein dynamisches Verhalten kann sich die Software der Auslastung anpassen und auch in kritischen Momenten noch genügend reagieren. Diesbezüglich mussten aber noch allgemeine Konzepte erarbeitet werden.

12.3 Vorgehen

Viele Arbeitsschritte werden von den Klassen des Frameworks bereits abgenommen. So ist es einfacher

- Objekte zu serialisieren
- Verteilte Applikationen zu schreiben (Skeleton – Stub)
- Zellen zu entwickeln
- XML einzusetzen

und vieles mehr.

Um mit dem Framework zu arbeiten sind gewisse Vorkenntnisse notwendig. So muss ein Entwickler die Programmiersprache C++ kennen. Für Java Entwickler ist der Einstieg nicht allzu schwierig, da viele Klassen und Konzepte von Java abgebildet wurden. So existieren ähnliche Konzepte für Streaming, Sockets und Threading. Es gibt aber komplett andere Konzepte zur Speicherverwaltung und natürlich die Zellenarchitektur. Diese müssen zuerst erlernt werden, wenn sich jemand mit dem Framework befassen möchte. Es braucht aber kein Ingenieur-Wissen, solange man nicht künstliche Intelligenz oder andere komplexe Konzepte einsetzen möchte.

13 Module Object Model Spezifikation 2.0

Die Module Object Model Spezifikation dient der allgemeinen Vereinbarung und der Definition von Objekten in der Frameworkumgebung. Das Zeus-Framework ist eine Referenzimplementation dieser Spezifikation.

Das Module Object Model (MOM)[2]definiert den Aufbau eines dynamisch und modularen Frameworks. Das MOM besteht primär aus einem Baum von Softwarekomponenten (Objekten), die dynamisch geladen werden können.

13.1 Eigenschaften des MOM

Das MOM hat durch seine einheitliche Entkopplung von Komponenten folgende Eigenschaften:

- Hohe Wiederverwendbarkeit der Komponenten. Das ermöglicht hervorragendes Software-Recycling.
- Dezentrale Entwicklung von Softwarekomponenten. Durch klare Schnittstellen können komplexe Systeme dezentral entwickelt werden.
- Stufenweises Entwickeln. Das System muss nicht schon beim Projektstart ins Detail geplant werden. Komponenten können auch später spezifiziert, entwickelt und integriert werden.
- Erweiterbarkeit der Software. Durch sich ändernde Kundenwünsche kann die Software auch später noch einfach erweitert werden.
- Durch die flexible Konfiguration kann die Software nach Kundenwunsch angepasst werden. Es können beliebige Komponenten in den Objektbaum eingehängt oder entfernt werden.
- Gute Wartbarkeit von komplexen Systemen. Einzelne Komponenten können ausgewechselt oder neu geschrieben werden, ohne dass das gesamte System neu bearbeitet werden muss.

2 MOM 1 wurde bei Studer.com definiert. [STWIN]

Das Modell hat auch seine Schwachpunkte, die aber durch klare Richtlinien bei der Entwicklung wieder wett gemacht werden können:

- Die Vernetzung der Komponenten ist nicht begrenzt. Legt ein Projektteam die Vernetzung nicht fest, kann ein System entstehen, welches sehr viele und unüberblickbare Abhängigkeiten besitzt.

13.2 Aufbau und Definition

Die MOM Spezifikation wird in 2 Anforderungskategorien unterteilt:

1. Die Basis-Anforderungen definieren die MUSS-Kriterien

2. Die erweiterten Anforderungen definieren Zusätze, welche implementiert werden können.

Jedes Objekt ist durch eine einheitliche Schnittstelle von den anderen Objekten entkoppelt. Es gibt 2 Arten von Objekttypen:

- *X-Objekt*: Das X-Objekt ist ein einfaches Objekt, welches eine Softwarekomponente repräsentiert. Das X-Objekt implementiert die Basis-Anforderungen der MOM Spezifikation.

- *Modul*: Das Modul ist eine Komponente welche eine ganze Software-Bibliothek repräsentiert. Das Modul implementiert die Erweiterungen der MOM Spezifikation. Zudem kann das Modul eine Session von einer Bibliothek erzeugen (Kontext). Die Modulsession ist ein Spezialfall und wird in diesem Dokument nicht weiter berücksichtigt.

13.2.1 Basis Anforderungen

Das MOM beschreibt sich folgendermassen:

- X-Objekte können Endknoten sein, welche im System eine Dienstleistung oder eine bestimmte Aufgabe erfüllen.

- X-Objekte können Knoten sein, welche andere X-Objekte zu logischen Einheiten gruppieren.

- X-Objekte können eingefroren oder freigeschaltet werden.
    - Beim Einfrieren der X-Objekte werden alle Referenzen freigegeben, die von anderen X-Objekten stammen. Eingefrorene X-Objekte reagieren nicht mehr.
    - Beim Freischalten können die benötigten Referenzen von anderen X-Objekten angefordert und gebraucht werden.
- Das MOM definiert den Datenaustausch zwischen X-Objekten durch vordefinierte Schnittstellen.
- Ein X-Objekt kann seine Kinderobjekte verzögert erstellen. Die Kinderobjekte werden erst erstellt, wenn jemand aus dem System ein solches Kinderobjekt referenzieren möchte.

13.2.2 Erweiterungen

Die Erweiterungen definieren spezielle X-Objekte, die *Module*:

- Module können aktiviert oder deaktiviert werden. In einem System ist immer nur ein Modul aktiv (Siehe Abbildung 30). Die Aktivierung wird vor allem bei graphischen Anwendungen verwendet. Das aktive Modul erhält hier den Eingabefokus.
- Module laden ihre Ressourcen (Shared Libraries, Dokumente oder stellen Verbindungen zu anderen Systemen her). Hat ein Modul, welches aktiviert oder von dem Objekte angefordert wurde, seine Ressourcen noch nicht geladen, erfolgt nun der Ladeprozess. Beim Deaktivieren bleiben die geladenen Ressourcen meistens erhalten.
- Manager sind Module die andere Module verwalten und zu logischen Gruppen zusammenfassen.
- Werden in einem Objektbaum Module und Manager eingesetzt, ist es wichtig, dass jedes Modul und jeder Manager ein Kindobjekt von einem Manager ist. Der Wurzelknoten muss also ein Manager sein. Nur so ist der Aktivierungsprozess nach Definition gewährleistet.

Jeder Manager und jedes Modul kann beliebig viele X-Objekte als Kinderobjekte besitzen.

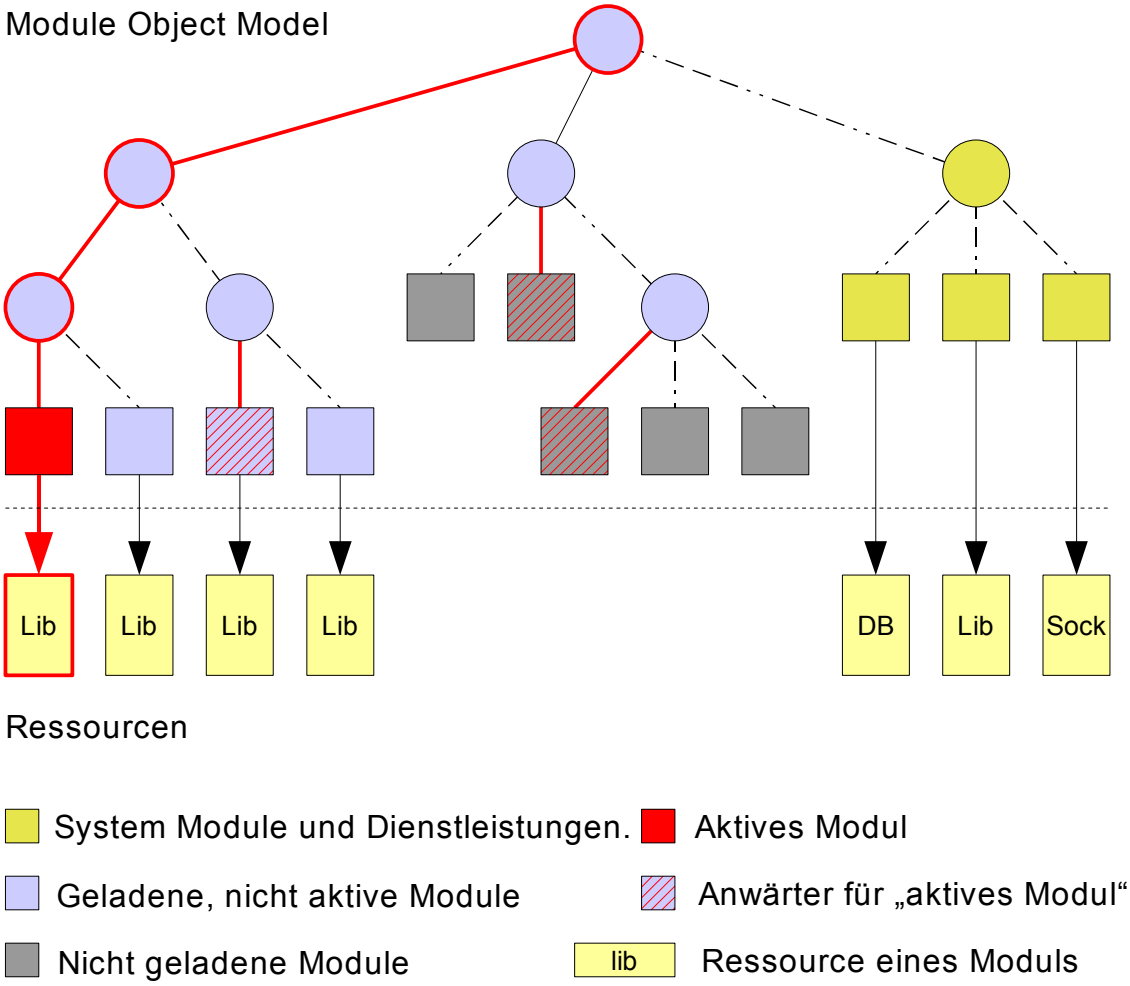


Abbildung 30: Diese Darstellung verdeutlicht den Aufbau und die Funktionsweise des MOM mit Modulen. Kreise symbolisieren die Manager, die Quadrate die Module. Das aktive Modul (rot) ist durch einen Aktivierungspfad gekennzeichnet. Die Systemmodule werden immer geladen. Sie stehen dem System generell zur Verfügung.

14 Das Klassen-Framework

Zur Entwicklung das Frameworks wurde ein eigenes Klassen-Framework aufgebaut. Es ermöglicht das Entwickeln und Betreiben von Applikationen in Linux und Windows. Das Klassen-Framework umfasst:

- Verschiedene Datentypen, wie Listen, Strings, Variant etc.
- Die MOM 2 Implementation
- Die Zellenarchitektur nach CCM
  - Threading
  - Networking
  - Messaging
  - Zellen
- Künstliche Intelligenz, wie genetische Algorithmen, Fuzzy Logic etc.
- Serialisierung von Objekten
  - Streaming
  - Objektfabriken
- Remote Method Invocation für C++

Das Framework kann beliebig erweitert werden. Es ist möglich ein eigenes Framework zu entwickeln, welches alle Eigenschaften des Zeus-Frameworks erbt. Die Klassen wurden in Paketen zusammengefasst und stehen als Klassen-Bibliothek (`zeusbase.lib` und `zeusdev.lib`) zur Verfügung.

- Die `zeusbase.lib` beinhaltet alle Basis-Klassen des Zeus-Frameworks.
- Die `zeusdev.lib` beinhaltet alle Hilfsklassen für einen Applikationsentwickler.

14.1 Vorschriften bei der Entwicklung mit Zeus

Das Zeus-Framework schreibt gewisse Vorgehensweisen bei der Entwicklung von Applikationen vor. Diese Vorschriften basieren einerseits auf Erfahrung und sind anderseits notwendig, um gewisse Plattformen zu unterstützen.

14.1.1 Schnittstellen

Schnittstellen sind ein wesentliches Konzept im Zeus-Framework. Jeder komplexere Datentyp (wie Strings, Listen etc.) braucht eine Schnittstelle. Auch jedes Objekt implementiert mindestens die Basisschnittstelle `IZUnknown`. Das hat folgende Gründe:

- Wenn die Objekte über Schnittstellen kommunizieren, sind die Abhängigkeiten relativ klein.
  - Die Wartung und Erweiterbarkeit von Software wird wesentlich besser.
  - Das System wird übersichtlicher.
  - Es vereinfacht das Zusammenarbeiten im Projektteam.
- Objekte können irgendwo in einem Speicher sein, sie können sich also auch in einen lokalen Heap befinden.
  - Das geschieht zum Beispiel, wenn eine Borland C++ Applikation eine Bibliothek lädt, die in Visual C++ entwickelt wurde. Die Applikation und die Bibliothek besitzen beide unterschiedliche Heaps. Die Verwaltung der Objekte darf nur vom entsprechenden Heap Organizer durchgeführt werden.
  - Manipulationen auf dem falschen Heap (wie `realloc`, `new` und `delete`) führen zu Problemen. Deshalb ist ein spätes Binden (late binding) notwendig, damit die Aktion im richtigen Heap ausgeführt wird.
- Eine gute Architektur bedient sich der Möglichkeit von Schnittstellen.
  - Trennen von Definition und Implementation.
  - Abstraktes Denken verringert Fehler im Design und Code.

Bei der Entwicklung von Schnittstellen in C++ ist auf folgendes zu achten:

- Alle Methoden sind abstrakt definiert.
- Schnittstellen dürfen keine Attribute besitzen. (Auch keine statische Attribute. Diese sind eh nicht global, sondern pro Modul einmal vorhanden. Besser Singletons verwenden!).
- Alle Übergabeparameter sind primitive Datentypen oder Schnittstellen. Es dürfen keine konkreten Objekte übergeben werden. Die Schnittstellen werden als Referenzen oder Zeiger übergeben.
- Wird von einer Schnittstelle eine andere Schnittstelle verlangt, muss der Referenzzähler um eins erhöht werden. Diese Schnittstelle darf nicht direkt als Rückgabewert zurückgegeben werden, da in C++ die Rückgabe ins Leere gehen kann (verursacht ein Ressourcen Leck).

Der letzte Punkt ist sehr wichtig. Alle angeforderten Schnittstellen müssen wieder freigegeben werden. Beispiel eines Schnittstellen-Benutzers:

Rückgabe von Schnittstellen

```
TString strName = L"./CellSystem1/MyCell";
IXObject* pMyCell = NULL;
if (pRoot->getObject(&strName, &pMyCell) == RET_NOERROR)
{
  ...
  pMyCell->release();
}
```

Tabelle 1: Bei der Rückgabe einer Schnittstelle wird der interne Referenzzähler automatisch erhöht. Deshalb muss ein `release()` aufgerufen werden. Wird dies nicht gemacht, bleibt das Objekt im Speicher und wird nie mehr gelöscht. Das könnte ein Systemabsturz auslösen.

Beispiel einer Implementation einer solchen Methode:

Rückgabe von Schnittstellen

```
unsigned long MQUALIFIER TTest::getObject(IString* pName, IXObject**
pObject)
{
  unsigned long ulRetval = RET_INVALID_PARAMETER;
  if (pName != NULL && pObject != NULL)
  {
    IXObject* pObjectToFind = this;

    //Try to find object
    ...

    //Return object
    *pObject =  pObjectToFind;
    (*pObject)->addRef();
    ulRetval = RET_NOERROR;
  }
  return ulRetval;
}
```

Tabelle 2: Wird bei solchen Methoden ein `RET_NOERROR` zurückgegeben, muss der interne Referenzzähler um eins erhöht werden. Bei jedem anderen Rückgabewert dürfen keine Referenzzähler erhöht werden.

14.1.2 Die Basisschnittstelle IZUnknown

Im Klassen-Framework sind alle Objekte von Schnittstellen abgeleitet. Die Basisschnittstelle heisst `IZUnknown` und regelt die folgenden Zugriffe:

- Verwaltung von Referenzen. Die Verwaltung regelt, wann ein konkretes Objekt gelöscht wird. Ein direktes `delete` führt zu Fehler und sollte vermieden werden.

- Casting zu anderen Schnittstellen. Da eine Art Mehrfachvererbung gebraucht wird, können durch direktes casting falsche Zeiger entstehen. Das geschieht vor allem beim Casten von einer beliebigen Schnittstelle zur anderen. Eine Ausnahme ist das Casting auf eine Basisklasse.

Die Methoden der Basisschnittstelle sind definiert als:

API	Zeusbase/System/Interfaces/IZUnknown
	`unsigned long MQUALIFIER askForInterface(long interface_id, IZUnknown** iface)` Diese Methode dient dem Abfragen und ggf. Casten auf eine andere Schnittstelle.
	`unsigned long MQUALIFIER addRef()` Diese Methode erhöht den internen Referenzzähler um eins.
	`unsigned long MQUALIFIER release()` Diese Methode verringert den internen Referenzzähler um eins. Fällt der Referenzzähler auf Null, wird das Objekt gelöscht.

Jede Schnittstelle hat eine eindeutige ID. Diese ID ist in der aktuellen Version des Zeus-Framework einer Nummer. Die Schnittstellen-IDs des Frameworks sind in der Datei `./zeusbase/Config/InterfaceID.hpp` definiert. Kundenspezifische Schnittstellen-IDs können in der Datei `./zeusbase/Config/CustomInterfaceID.hpp` definiert werden.

Die Rückgabewerte, wie zum Beispiel `RET_NOERROR`, sind ebenfalls in der Datei `./zeusbase/Config/InterfaceID.hpp` definiert.

Das folgende Beispiel zeigt ein Casting von einer Basisschnittstelle zu einer spezialisierten Schnittstelle.

Casting mit Schnittstellen

```
IXRootObject* pRoot = NULL;
if (pObject->askForInterface(INTERFACE_IXRootObject,
(IZUnknown**)&pRoot) == RET_NOERROR)
{
  ...
  pRoot->release();
}
```

Tabelle 3: Beim erfolgreichen Casting wird `RET_NOERROR` zurückgegeben. Der interne Referenzzähler wird automatisch erhöht, deshalb muss danach ein `release()` erfolgen.

14.1.3 Die Basisklasse TZObject

Alle konkreten Klassen sind von dieser Basisklasse abgeleitet. Sie implementiert die Methoden der Basisschnittstelle `IZUnknown`.

Will eine solche Klasse weitere Schnittstellen implementieren, geschieht dies in C++ durch die Mehrfachvererbung. Durch vorgefertigte Makros wird die Speicherverwaltung automatisch in die Klasse eingefügt. Die Makros sind in der Datei `./zeusbase/System/Interfaces/IZUnknownImplHelper.hpp` definiert.

API	`MEMORY_MANAGER_DECL;`
	Dieses Makro deklariert die Speicherverwaltung im Header der Klassendefinition.

API	`MEMORY_MANAGER_IMPL(classid);`
	Dieses Makro startet die Implementation der Speicherverwaltung. Es gibt eine Variante mit Locks, um eine thread-sichere Speicherverwaltung zu implementieren.

API	`INTERFACE_CAST(interface_type, interface_id);`
	Dieses Makro definiert ein speziellen Cast. Es muss der Typ der Schnittstelle angegeben werden und die eindeutige Nummer der Schnittstelle (ID).

API	`MEMORY_MANAGER_IMPL_END;`
	Beendet die Implementation der Speicherverwaltung. Es gibt die Variante der Delegation aller Aufrufe an eine spezielle Vaterklasse. Die Vaterklasse `TZObject` wird standardmässig gebraucht.

Folgende Bespiele illustrieren das Definieren einer solchen Klasse:

Klassendefinition in Zeus

```
class TTest : public TZObject, public ITest
{
  public:
    TTest();

    //Methods of ITest
    virtual unsigned long MQUALIFIER getObject(IString* pName,
                                               IXObject** pObject);
    ...
    //Methods of IZUnknown
    MEMORY_MANAGER_DECL

  protected:
    virtual ~TTest();
    ...
  private:
    ...
};
```

Tabelle 4: Die Mehrfachvererbung besteht darin, zuerst von einer Hilfsklasse oder Vaterklasse und dann von Schnittstellen zu erben.

Implementation einer Klasse

```
TTest::TTest() : public TZObject()
{...}

...

//Implementation der Speicherverwaltung
MEMORY_MANAGER_IMPL(TTest);
  INTERFACE_CAST(ITest, INTERFACE_ITest);
MEMORY_MANAGER_IMPL_END;
```

Tabelle 5: Bei der Implementation der Speicherverwaltung können Makros verwendet werden.

14.2 Datentypen des Zeus-Frameworks

Das Framework bietet verschiedenste Datentypen an, wie zum Bsp.:

API	
`TString`	Unicode fähiger String, welcher die Schnittstelle `IString` implementiert.
`TSingleLinkedList`	Template-Klasse, welche Objekte in einer verketteten Liste verbindet. Implementiert die Schnittstelle `IList`.
`TArrayList`	Eine Template-Klasse, welche ein Array verwaltet. Implementiert die Schnittstelle `IList`.
`TByteArray`	Diese Klasse verwaltet ein char-Array und wird für Streaming-Anwendungen gebraucht.
`TStringList`	Eine Liste für Strings vom Typ `TString`. Diese Liste implementiert die Schnittstelle `IList<IString*>`.
`TStringMap`	Ein binärer Suchbaum, welcher als Schlüssel einen String-Wert akzeptiert.
`TStack`	Eine Template-Klasse, die einen Stack implementiert.
`TQueue`	Eine Template-Klasse, die eine Queue implementiert.
`TZVariant`	Ein Variant zum Kapseln von anderen Datentypen und Objekten.

Die Datentypen erweitern die primitiven Datentypen. Von den primitiven Datentypen werden folgende vollständig unterstützt:

API	
`long`	Ganzzahliger Wert mit Vorzeichen (32 Bit)

API	
`unsigned long`	Ganzzahliger Wert ohne Vorzeichen (32 Bit)
`bool`	Boolescher Wert (32 Bit)
`double`	Reeller Zahlenwert (64 Bit)
`char`	Ganzzahliger Wert mit Vorzeichen (8 Bit)
`unsigned char`	Ganzzahliger Wert ohne Vorzeichen (8 Bit)
`wchar_t`	Wide Character für UNICODE

Das Zeus-Framework stellt eine umfassende Funktionalität mit XML[3] Daten zur Verfügung. Lesen, Schreiben und Traversieren von XML Knoten sind die Grundfunktionen.

API	
`IXMLParser`	Schnittstelle zum Lesen eines XML Dokuments
`IXMLDocument`	Verwaltungsklasse eines XML Dokuments
`IXMLNode`	Ein XML Knoten, welcher Attribute und Text beinhalten kann.
`IXMLNodeList`	Eine Liste von XML Knoten.

14.3 Klassen zum MOM 2

Die MOM Struktur wird in dieser Implementation als XML-Baum eingelesen

3 In dieser Version von Zeus-Framework wird der Parser von Apache verwendet. [XML02]

und dynamisch aufgebaut. Ein X-Objekt ist ein Objekt, welches seine Daten (Attribute) in einem XML Knoten lesen und speichern kann.

Ein XML Knoten kann Kinderknoten besitzen. Die Kinderknoten können auch wieder X-Objekte darstellen. Diese X-Objekte werden automatisch dem übergeordneten X-Objekt als Kinderobjekte untergeordnet.

Eine Objektfabrik versucht aus einem XML Dokument von jedem Knoten ein X-Objekt zu erzeugen und so einen Objektbaum aufzubauen. Jeder XML Knoten muss deshalb spezifizieren, welche Klasse instanziert werden soll, damit die Fabrik das entsprechende Objekt erzeugen und am richtigen Ort im Baum platzieren kann.

Als Basisklasse eines X-Objekts wird das `TXObject` definiert. Diese Klasse verwaltet eine Liste von Kinderobjekten, welche ebenfalls vom Typ `TXObject` sind. Durch diese rekursive Formulierung wird der Objektbaum definiert.

Die Basisklasse wird erweitert durch die Klassen:

- `TXRootObject`. Definiert den Wurzelknoten des Objektbaums.
- `TXLoaderObject`. Definiert einen Knoten, welcher folgende Funktionalität implementiert:
  - Kann verhindern, dass beim Aufbauen des Objektbaums ein Unterbaum erstellt wird. Dieser Unterbaum wird später erst bei Gebrauch erstellt.
  - Kann ein Unterbaum aus einem anderen XML Dokument laden. Dies ermöglicht eine Modularisierung der XML Dokumente

Diese Klassen implementieren die Basisanforderungen der MOM Spezifikation. Sie dienen weiter als Persistenzlayer. Beim Starten der Applikation können X-Objekte ihre Einstellung (Attribute) aus der XML Struktur lesen und beim Beenden können sie ihre Attribute wieder in die XML Struktur speichern. Diese XML Struktur kann in einer Datenbank (XML fähige Datenbank) oder als XML-Datei gespeichert werden.

Weitere Informationen zur XML Definition von X-Objekten siehe Kapitel auf Seite .

API	
`IXObject`	Basisschnittstelle des MOM 2.
`IXRootObject`	Schnittstelle für das Wurzelobjekt.

API	
`IXLoaderObject`	Schnittstelle für ein ladbares X-Objekt
`TXObject`	Implementation eines X-Objekts. Andere X-Objekte können von dieser Klasse erben.
`TXRootObject`	Implementation eines Wurzelobjekts. Andere Wurzelobjekte können von dieser Klasse erben.
`TXLoaderObject`	Implementation eines ladbares X-Objekts.
`TXObjectFactory`	Fabrik welche X-Objekte aus einem XML Dokument erstellen kann.

Die erweiterten Eigenschaften des MOM 2 wurden durch die Klassen `TModule` und `TSystemManager` implementiert. Die Basisklasse des Moduls heisst `TModule`. Sie implementiert die Schnittstelle `IModule`, welches das Aktivieren und deaktivieren von Modulen spezifiziert.

Wichtig:
Damit das Aktivieren, bzw. Deaktivieren von Module restlos funktioniert, muss darauf geachtet werden, dass die Objekthierarchie geschlossen aus Objekten besteht, welche den Typ `IModule` vererben. Alle anderen Objekte können nicht aktiviert, bzw. deaktiviert werden.

Die Klasse `TSystemManager` ist vom Typ `TXRootObject` abgeleitet und implementiert ebenfalls die Schnittstelle `IModule`. Der Systemmanager ist das Wurzelobjekt im Modul-Objektbaum.

API	
`IModule`	Basisschnittstelle aller Module und Manager.
`TModule`	Implementation des Moduls und des Managers. Das Modul kann auch die Aufgabe des Managers übernehmen.
`TSystemManager`	Implementation eines Wurzelobjekts in Form eines Moduls.

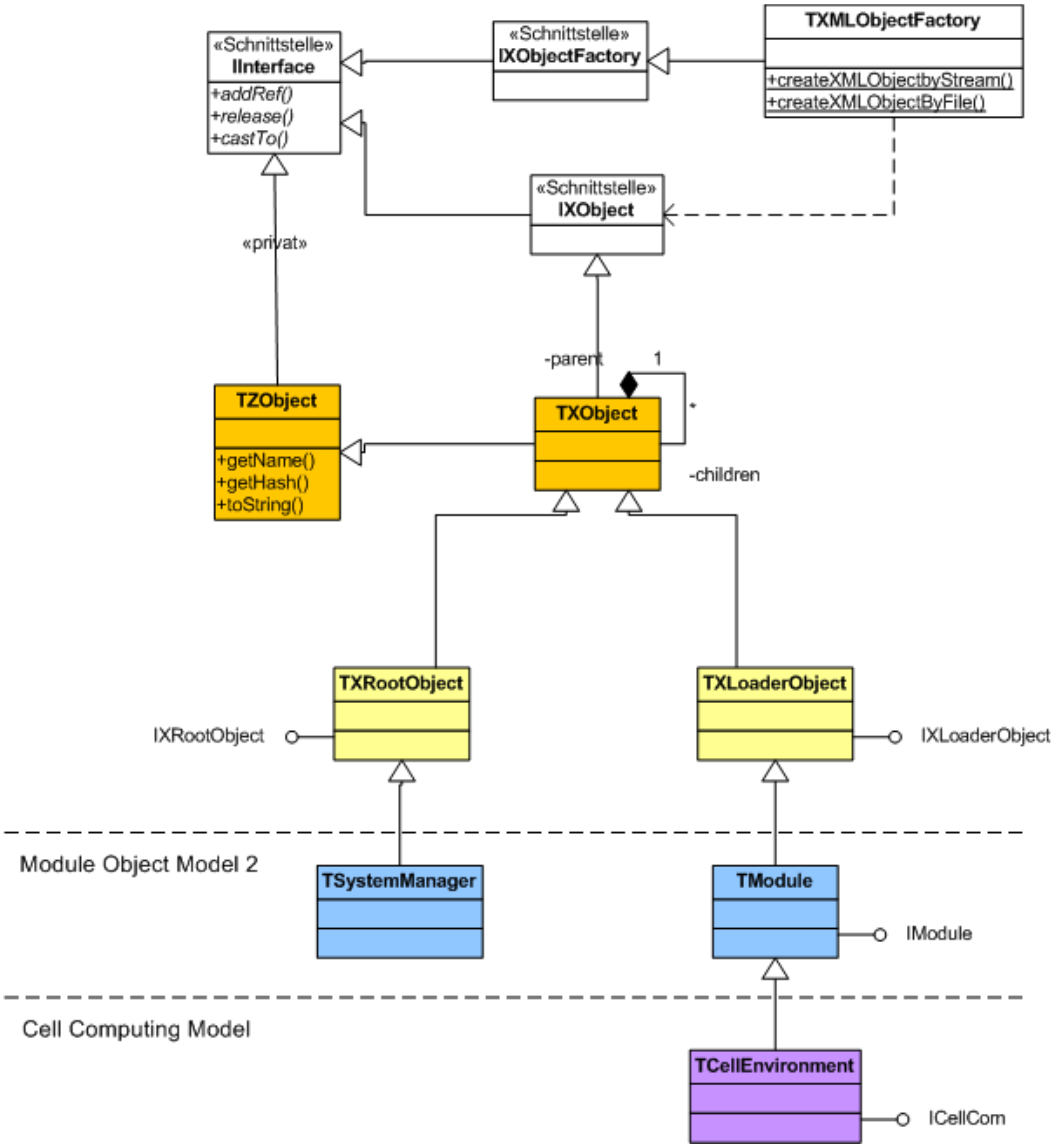


Abbildung 31: Basisklassen und Ableitungen

14.3.1 Implementation eines X-Objekts

Ein X-Objekt erbt immer von einer Basisklasse `TXObject` oder `TXRootObject`, oder von einer Unterklasse dieser beiden Basisklassen.

Zur Registrierung von X-Objekten wurden Makros angefertigt. Die Makros sind in der Datei `zeusbase/System/Interfaces/IXObjectImplHelper.hpp` definiert.

API	`REG_SUB_BEGIN(classid);`
	Dieses Makro startet die Definition der X-Objekt Registrierung. Als Parameter muss die ID der Klasse angegeben werden.

API	`REG_SUB_ADD(classid, classname);`
	Dieses Makro dient dem Zuordnen eines Klassennamens zu der ID der Klasse. Ein X-Objekt kann auch mehrere Klassennamen besitzen (im XML), welche die gleiche ID referenzieren.

API	`REG_SUB_END;`
	Dieses Makro beendet die Definition.

API	`REG_ROOT_BEGIN(classid);`
	Dieses Makro startet die Definition der X-Wurzelobjekt Registrierung. Als Parameter muss die ID der Klasse angegeben werden.

API	`REG_ROOT_ADD(classid, classname);`
	Dieses Makro dient dem Zuordnen eines Klassennamens zu der ID der Klasse. Ein X-Objekt kann auch mehrere Klassennamen besitzen (im XML), welche die gleiche ID referenzieren.

API	`REG_ROOT_END;`
	Dieses Makro beendet die Definition.

Die X-Objekte können in so genannten Code-Modulen (Bibliotheken) entwickelt und für eine Anwendung bereitgestellt werden. Damit die Objektfabrik die Klasse eines X-Objekts auch findet und ein Objekt instanziern

kann, muss die Klasse entweder registriert werden, oder sie wird durch eine Fabrikfunktion in einem Code Modul exportiert. Informationen zur Programmierung von externen Code Modulen sind im Kapitel [] Seite zu finden.

Das direkte Registrieren bei der Objektfabrik wird durch ein entsprechendes Makro getätigt. Dieses Makro muss vor dem Laden des Objektbaums ausgeführt werden.

API	`XOBJECTFACTORY_REGISTER_ROOT(classid);`
	Dieses Makro registriert ein Wurzelobjekt bei der Objektfabrik.

API	`XOBJECTFACTORY_REGISTER_SUB(classid);`
	Dieses Makro registriert ein X-Objekt bei der Objektfabrik.

Es gibt auch Makros zum Abmelden von X-Objekten

API	`XOBJECTFACTORY_UNREGISTER_ROOT(classid);`
	Dieses Makro meldet ein Wurzelobjekt bei der Objektfabrik ab.

API	`XOBJECTFACTORY_UNREGISTER_SUB(classid);`
	Dieses Makro meldet ein X-Objekt bei der Objektfabrik ab.

Das folgende Beispiel zeigt eine Definition der X-Objektklasse `TInputCell`. Eine Wurzelobjekt wird mit den entsprechenden Makros analog definiert.

InputCell.h

```
#include <zeusdev/AbstractCell.h>
#include <zeusbase/System/XObjectFactory.h>

BEGIN_NAMESPACE_Zeus

class TInputCell : public TAbstractCell
{
  public:
    TInputCell(IXMLNode* node);

    REG_SUB_BEGIN(TInputCell)
    REG_SUB_ADD(TInputCell, L"TInputCell");
    REG_SUB_END
...

  protected:
    virtual ~TInputCell();

  private:
...
};

END_NAMESPACE_Zeus
```

Tabelle 6: Header Datei für das X-Objekt TInputCell.

14.3.2 Wichtige Methoden des X-Objekts

Ein paar Methoden sind hier zu erwähnen, die bei der Implementation von X-Objekten beachtet werden müssen.

API	`Zeusbase/System/XObject`
	`bool MQUALIFIER unfreeze()` Diese Methode wird bei allen X-Objekten aufgerufen, wenn der Objektbaum vollständig geladen wurde. Ein X-Objekt soll erst hier seine Referenzen im Objektbaum ermitteln. Diese Methode muss der Basisklasse weitergeleitet werden!
	`bool MQUALIFIER freeze()` Diese Methode wird jedem X-Objekt aufgerufen, wenn der Objektbaum freigegeben werden soll. Diese Methode muss der Basisklasse weitergeleitet werden!

API	`Zeusbase/System/XObject`
	`void onChildAdded(IXObject* child)` Diese callback Methode wird beim Hinzufügen eines Kinderobjekts aufgerufen.
	`void onChildDeleted(IXObject* child)` Diese callback Methode wird beim Entfernen eines Kinderobjekts aufgerufen.
	`unsigned long MQUALIFIER getObject(IString* path, IXObject** obj);` Mit dieser Methode können andere Objekte im Objektbaum ermittelt werden. Der Pfad wird im folgenden Kapitel beschrieben.
	`unsigned long MQUALIFIER getRootObject(IXObject** obj)` Ermittelt das Wurzelobjekt des Objektbaums.

14.3.3 Objektpfad

Der Objektpfad dient dem Suchen und Ermitteln von X-Objekten in einem geladenen Objektbaum. Der Pfad ist aufgebaut wie ein Verzeichnispfad bei Unix. Er enthält folgende Elemente:

API	`zeusbase/System/XObjectPath`
`/`	Repräsentiert das Wurzelobjekt
`./`	Referenziert das aktuelle Objekt (this)
`../`	Referenziert das Vaterobjekt
`./MyChild`	Referenziert das Kindobjekt mit dem Namen `MyChild`
`./MyChild[2]`	Referenziert das dritte Kindobjekt mit dem Namen `MyChild`

Diese Elemente des Pfads können nach belieben zusammengesetzt werden.

14.4 Die Zellenarchitektur

Durch die Zellenarchitektur erhält das System seine Dynamik, während mit X-Objekten (siehe oben) die statische Struktur festgelegt wird. Zur Implementation der Zellen sind weitere Konzepte und Verfahren nötig, welche in diesem Kapitel mit den dazugehörigen Klassen beschrieben werden.

14.4.1 Die Zellen

Für das Zeus-Framework wurde eine Zellenumgebung (Klasse `TCellEnvironment`) entwickelt, welche eine beliebige Zellenart beherbergen kann (Abbildung 31). Die Zellenumgebung erbt von der MOM Klasse `TModule` und übernimmt die Kontroll- und Steuerfunktion eines bestimmten Zellensystems:

- Verwalten der Kommunikationskanäle
- Überwachen der Zellen
- Vermehren der Zellen
- Abbauen von Zellen

Die entsprechenden Zellen werden in Code-Modulen (Bibliotheken) entwickelt. Diese werden zur Laufzeit vom Framework dynamisch geladen damit die Zellenobjekte erzeugt werden können.

API	
`ICellKernel`	Schnittstelle für den Zellkern. Diese muss von einem Applikationsentwickler implementiert werden.
`ICellComm`	Schnittstelle des Zellmantels. Durch diese Schnittstelle können Meldungen aus der Zellenumgebung gelesen oder Meldungen abgesetzt werden.

API	
`ICell`	Interne Schnittstelle für die Diagnose von Zellen. Sie repräsentiert den äusseren Zellmantel.
`ICellSurvey`	Schnittstelle der Überwachungszelle.
`TCellEnvironment`	Umgebung einer oder mehreren Zellen. Die Umgebung erstellt Kommunikationskanäle und registriert die Zellen für lokale oder entfernte Kommunikation.
`TCell`	Diese Klasse repräsentiert eine Zelleninstanz. Sie stellt Diagnosemöglichkeiten zur Verfügung. Sie besitzt die Fähigkeit des Klonens.
`TCellSurvey`	Diese Zelle ist spezialisiert, um ein Zellensystem zu überwachen und Daten von Zellen auszuwerten.
`TAbstractCell`	Hilfsklasse zur Implementation eines Zellkerns. Dieser Zellkern wird vom Applikationsentwickler geschrieben.
`TRelyCell`	Umleitung von Meldungen. Diese Zelle kann als Gateway verwendet werden, um Meldungen zu filtern.

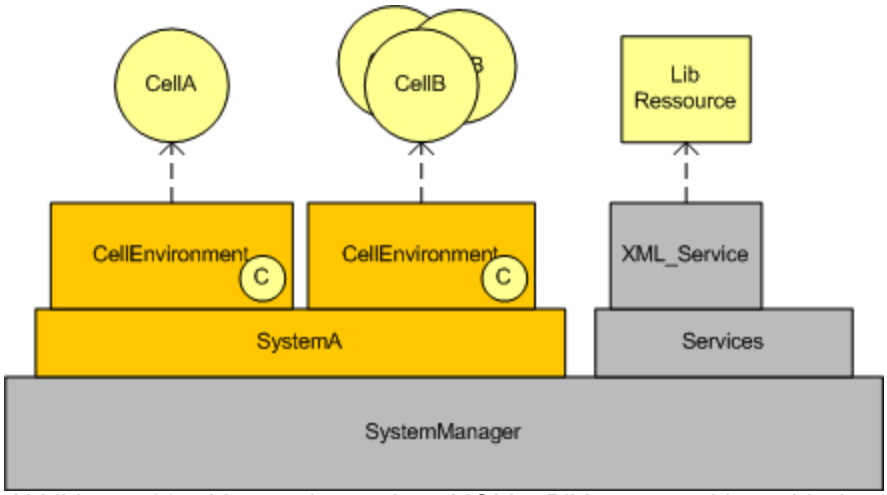


Abbildung 32: Verwendung des MOM. Bilden von hierarchischen Zellensystemen

Das `TCellEnvironment` implementiert die Möglichkeit Zellen zu klonen, und abzubauen. Diese 2 Verfahren sind im Dokument [CCM1] beschrieben.

14.4.2 Threading

Damit die Zellen parallel arbeiten können, brauchen es Threads. Fürs Threading unter Linux[4] und Windows wurden Klassen erstellt, die Funktionen und Datentypen der Betriebssystem-API kapseln.

API	
`TThread`	Ein Thread kann mit Hilfe dieser Klasse implementiert werden.
`TCriticalSection`	Kritische Abschnitte und Locks können mit dieser Klasse umgesetzt werden.
`TEvent`	Diese Klasse dient dem Signalisieren von Threads.
`TTime`	Klasse für Zeit-Funktionen, wie `sleep()` und

4 für Linux werden POSIX Threads eingesetzt [PTHREAD] und [POSIXTH]

API	
	`getTimeStamp()`.

14.4.3 Networking

Zellen müssen untereinander kommunizieren können. Zellen die im gleichen Prozess arbeiten, können direkt über den Shared Memory kommunizieren. Befinden sich die Zellen oder Zellensysteme auf entfernten Computersystemen, müssen sie über ein Netzwerk kommunizieren können. Diese Kommunikation wurde mit Klassen implementiert, welche Netzwerkdienste ansprechen.

Im Framework gibt es Klassen, die sich mit dem Networking befassen:

API	
`TSocket`	Diese Klasse implementiert einen Client Socket um TCP/IP Verbindungen aufzubauen
`TServerSocket`	Der Server kann mit dieser Klasse implementiert werden
`TNetworkInterface`	Ermittelt die Netzwerkschnittstellen mit den konfigurierten TCP/IP Adressen

14.4.4 Cell Communication Transfer Protocol

Mit dem Cell Communication Transfer Protocol CCTP können binäre Datenpakete ausgetauscht werden. Das Protokoll definiert einen Header von 12 Bytes für die Response- und Request-Pakete. Das Protokoll wird vor allem für die Kommunikation zwischen zwei Zellen verwendet.

Die Request-Pakete werden zum Senden der Datenpakete verwendet. Der Request dient der Anfrage beim entfernten System, ob sie die Meldung akzeptiert. Das entfernte System antwortet mit einer Response über aufgetretene Fehler oder Erfolge.

API	
`TCCTPRequest`	Request Objekt für das binäre Cell Communication Transfer Protocol
`TCCTPResponse`	Response Objekt für das CCTP

Der Body des Protokolls kann ein beliebiger binärer Datenblock sein, wie zum Bsp. Bilder, serialisierbare Objekte usw.

14.4.4.1 Request Pakete

Das Request Paket besteht aus folgenden Felder:

- Version: Dieses Feld dient der Protokollversion
- Methode: Definiert die Art und Weise wie der Request verarbeitet werden soll.
- Identität der Meldung: Diese ID kann wichtig sein, wenn Meldungen wiederholt werden müssen.
- Länge des Body: Anzahl Bytes des folgenden Datenblocks

Byte 1	Byte 2	Byte 3	Byte 4
Version	Methode	Reserviert	
Identität der Meldung (ID)			
Länge des Body in Bytes			
Body			

Tabelle 7: Format des Request-Pakets

14.4.4.2 Response Pakete

Das Response Paket besteht aus folgenden Teilen:

- Version: Protokollversion
- Zustand: Antwort, bzw Fehlercode (Analog zu HTTP)
- Länge des Body: Anzahl Bytes des folgenden Datenblocks

Byte 1	Byte 2	Byte 3	Byte 4
Version	Reserviert		
Zustand der Antwort			
Länge des Body in Bytes			
Body			

Tabelle 8: Format des Response-Pakets

14.4.5 Messaging

Zum Kommunizieren zwischen Zellen und Zellensystemen wurde ein Meldungssystem entwickelt. Die Meldungen können durch so genannte Kommunikationskanäle (Pipes) abgesetzt werden. Ein Kanal ist eine Verbindung von einer Zelle zu einer anderen Zelle oder zu einem Zellensystem. Das andere Ende des Kanals kann auf dem gleichen PC oder auf einem entfernten Rechner liegen.

Die Meldungsklassen sind alle von der Schnittstelle `ISerializable` abgeleitet und können so serialisiert übers Netzwerk gesendet werden. Dadurch lassen sich die Systemgrenzen einfach überwinden.

API	
`IMessage`	Basisschnittstelle für Meldungen.

API	
`IBinaryMessage`	Schnittstelle für Meldungen mit binärem Format.
`IXMLMessage`	Schnittstelle für Meldungen mit XML Inhalt.
`IObjectMessage`	Schnittstelle für Meldungen, die ein Objekt beinhalten.
`ISystemMessage`	Schnittstelle für Meldungen, die Systemdaten übermitteln.
`TBinaryMessage`	Klasse welche ein binäre Meldung repräsentiert.
`TXMLMessage`	Klasse welche eine XML Meldung repräsentiert.
`TObjectMessage`	Klasse welche eine Objekt Meldung repräsentiert.
`TSystemMessage`	Klasse welche eine Systemmeldung repräsentiert.

Mit diesen Klassen können nun verschiedenartige Meldungen realisiert werden:

- Kontrollmeldungen, wie Ping, Status Informationen etc.
- Die Arbeitspakete, welche Daten mit Prozessbeschreibung enthalten.
- Autonome Zellen (Meldungszellen, Agenten)

14.4.5.1 Die Kontrollmeldungen

Die Kontrollmeldungen dienen dem Steuern und Überwachen von Zellen und Zellensystemen. Diese Meldungen werden meistens von der Kontrollstruktur innerhalb eines Zellensystem verwendet (Überwachungszelle). Sie kann aber auch von externen Zellen zur Prüfung von Qualität und Lebenserhaltung verwendet werden.

Die berühmteste Meldung ist das Ping, welches prüft, ob der Kommunikationspartner noch lebt.

14.4.5.2 Die Arbeitspakete

Die Arbeitspakete enthalten Daten. Diese Daten können binär oder als Text übergeben werden.

➔ Extension: Arbeitspakete mit Workflow

Der Workflow ist in Form eines Petrinetzes angeben und wird jedem Arbeitspaket mitgegeben. In diesem Petrinetz wird gespeichert, wo sich das Paket zur Zeit befindet, ob die Zelle auf weitere Arbeitspakete warten muss und an welche Zelle(n) das bearbeitete Paket gesendet werden soll.

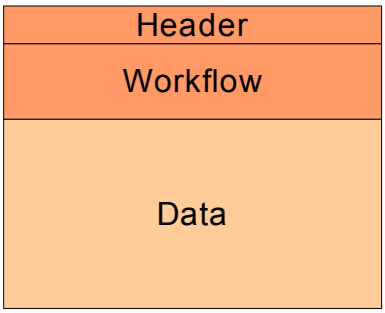


Abbildung 33: Aufbau der Arbeitspakete

14.4.6 Kommunikationskanäle

Will eine Zelle mit einer anderen Zelle kommunizieren, muss zuerst ein Kommunikationskanal erstellt werden. Durch diesen Kanal werden dann Arbeitspakete und Kommandos gesendet.

Das Zellenumgebung verwaltet alle Kommunikationskanäle, durch welche die Zellen zu Informationen gelangen können (Input Pipes). Diese Art von Kanälen werden in einer lokalen und einer globalen Registrierungsstelle (Namensdienst) registriert, damit andere Zellen Meldungen an dieses Zellensystem senden kann.

Die Zellenumgebung holt sich die ausgehenden Kommunikationskanäle

(Output Pipes) von den Registrierungsstellen. Aus Gründen der Effizienz können diese ausgehenden Kanäle auch in der Zellenumgebung zwischengespeichert werden. Das Suchen anderer Kommunikationskanäle wird mehrstufig durchgeführt:

1. Zuerst wird versucht die Zelle lokal zu finden. Das geschieht durch das Suchen einer Zelle im Framework mit Hilfe der lokalen Registrierungsstelle. Wird die Zelle gefunden, wird ein interner Kommunikationskanal erstellt (über den Shared Memory).

2. Wird die Zelle nicht gefunden, wird eine Registrierungsstelle im LAN (Local Area Network) abgefragt. Unter Umständen gibt es mehrere Zellen mit dem gleichen Namen, die aber auf verschiedenen Host-Rechner laufen. Die Registrierungsstelle gibt alle gefunden Zellen zurück, die in Frage kommen könnten. Die Anzahl der Rückgaben kann eingeschränkt werden, wenn viele gleichnamige Zellensysteme eingesetzt werden.

3. Werden entfernte Zellen gefunden, kann mit jeder oder einer gewissen Anzahl der Zellen eine Verbindung hergestellt werden (Netzwerk-Kommunikationskanal).

So kann zum Beispiel die Zelle A mit mehreren Zellen vom Typ B kommunizieren. Ein Arbeitspaket wird aber immer nur an eine bestimmte Zelle des Typs B gesendet. Dabei ist es Zufall, welche Zelle B das Paket erhält.

Alle Zellen einer Umgebung teilen sich die gleichen Eingabekanäle. Stirb eine Zelle ab, dann übernehmen andere Zellen dieser Umgebung die Arbeit.

Wenn die ganze Umgebung stirbt, sterben auch alle eingehenden Kommunikationskanäle (Input Pipes) dieser Zellenumgebung ab. Die verwaisten Kommunikationskanäle werden von anderen Zellen als `broken` erkannt und automatisch entfernt. Die ausgehenden Kommunikationskanäle (Output Pipes) werden nicht verändert, da sie nicht dieser Zelle gehören, sondern nur als sendende Schicht verwendet werden.

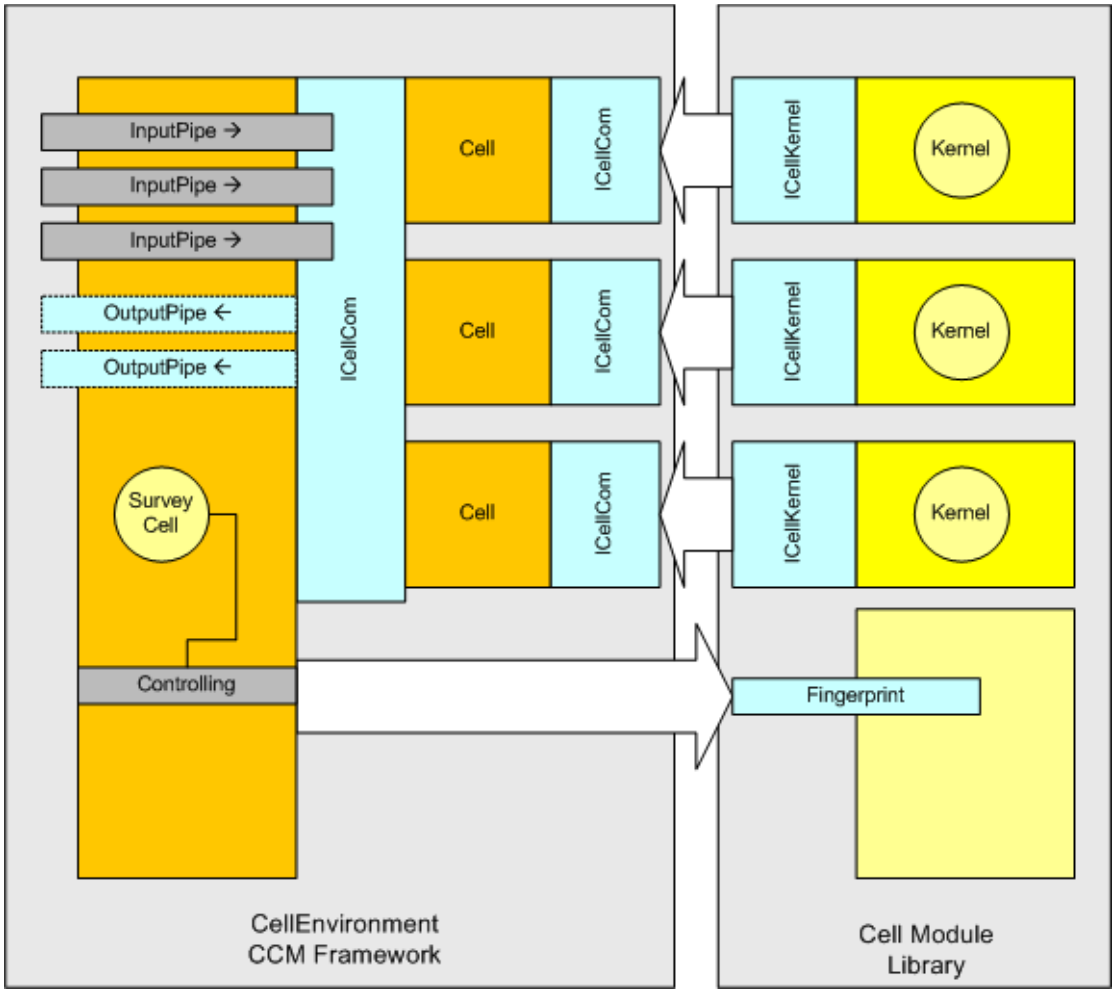


Abbildung 34: Die Bibliothek (Shared Library rechts) wird durch das Framework (links) geladen und verwaltet. Die Zellkerne befinden sich in der Bibliothek. Die Verwaltung der Kommunikationskanäle und der Zellen wird durch das Framework betätigt.

API	
`ICommPipe`	Schnittstelle eines Kommunikationskanals. Durch diese Schnittstelle können Meldungen abgesetzt werden
`IOwnCommPipe`	Dieses Schnittstelle eines Kommunikationskanals ist nur für den Besitzer eines Kanals verfügbar und dient dem Lesen von Meldungen.

API	
`ILocalPipeRegistry`	Schnittstelle der Registrierungsstelle für lokale und entfernte Kommunikationskanäle.
`TCommPipe`	Kommunikationskanal zum Senden und Empfangen von Meldungen.
`TLocalPipeRegistry`	Lokale Registrierungsstelle für Kommunikationskanäle. Dadurch können Kanäle lokal von anderen Zellen gefunden werden (lookup). Die Kanäle können aber auch im Namensdienst global registriert werden.

Durch den zweiten Mechanismus, dem Erkennen von neuen Kommunikationspartner, wird das dynamische Kommunikationsverhalten geschlossen. Stellt eine Zelle A mit Zellen des Typs B einer Verbindung her, kann es sein, dass zu einem späteren Zeitpunkt weitere Zellen des Typs B hinzu kommen. Natürlich ist es wünschenswert, wenn die Zelle A nun auch mit diesen neuen Zellen kommunizieren kann, automatisch versteht sich. Dazu wird nach einer bestimmten Zeit T die Registrierungsstelle erneut nach möglichen Zellen B abgefragt. Die neuen Zellen werden dann als mögliche Kommunikationspartner in der Zelle A registriert.

14.4.6.1 Lokale Kommunikationskanäle

Lokale Kommunikationskanäle sind intern im Framework implementiert. Es gibt unterschiedliche Arten der Kommunikation:

- Queuing: Die Arbeitspakete werden in eine Queue abgelegt. Für Queues gibt es keine Grenze der Anzahl Einträge. Der Klon-Algorithmus regelt die Auslastung der Queues eines Zellensystems (Siehe Kapitel zum Klonen).

- Queuing with Replace: Werden viele gleiche Pakete übertragen, interessieren aber nur die aktuellsten Pakete, kann das Queuing with Replace angewandt werden. Jedes Arbeitspaket besitzt eine ID. Wird die gleiche ID nochmals verwendet, und ist das vorhergehende Arbeitspaket noch in der Queue, wird dieses durch das aktuelle Arbeitspaket ersetzt. Der Vorteil ist, dass die Grösse der Queue klein gehalten werden kann. Dieses Verfahren wird vor allem bei der Signal-Visualisierung verwendet, um schnell auf Änderungen reagieren zu können und das Schleppen von Werten zu umgehen (Stau in der Queue). Dieses Verfahren steht in der aktuellen Version noch nicht zur Verfügung.

Ein lokaler Kommunikationskanal wird durch folgende Kenngrössen beschrieben:

- Name der Zelle
- Name des Kommunikationskanals (Pipe)

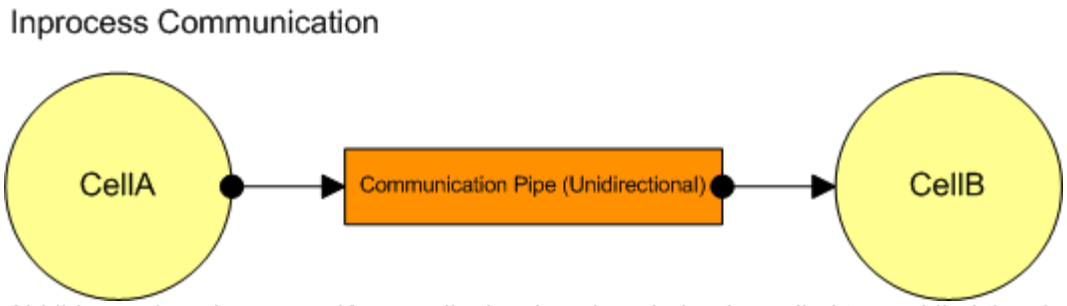


Abbildung 35: Inprozess-Kommunikationskanäle sind eine direkte unidirektionale Verbindung zwischen 2 Zellen

14.4.6.2 Entfernte Kommunikationskanäle

Damit Zellen über ein Netzwerk miteinander kommunizieren können, braucht es eine Registrierung. Folgende Daten der Zellen werden dort verwaltet:

- Name der Zelle
- Name des Kommunikationskanals (Pipe)

- Adresse des Host (Namen des Computers oder IP Adresse) (TCP/IP)
- Port des Socket (TCP/IP)

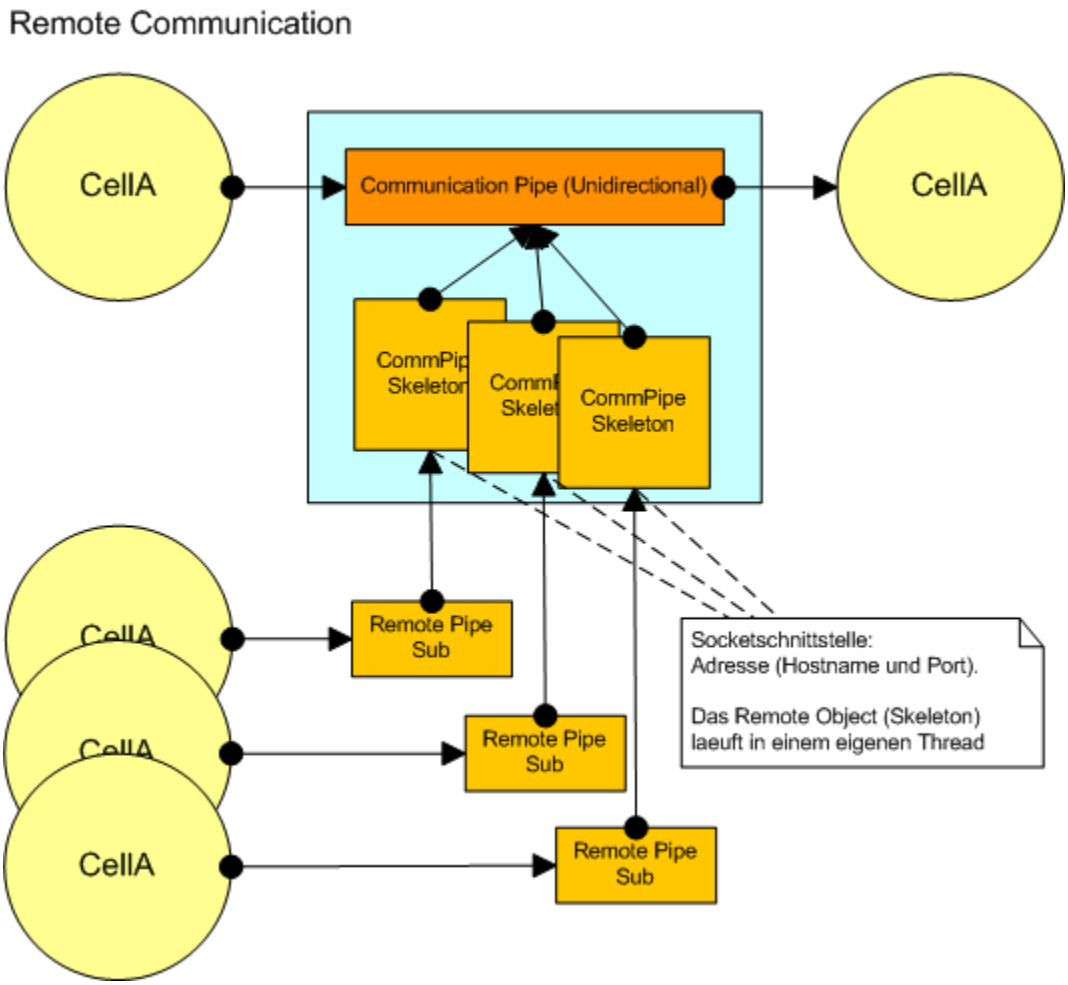


Abbildung 36: Entfernte Kommunikationskanäle werden mit dem Muster des Skeleton-Stub implementiert.

Die Registrierungsstelle ist eine eigene Applikation, welche für die laufenden Zeus-Applikationen zugänglich ist. Beim Starten einer Zelle werden ihre Kommunikationskanäle automatisch bei der Registrierungsstelle registriert.

Als Kommunikationsprotokoll wird das Cell Communication Transfer Protocol CCTP verwendet. Es handelt sich um ein binäres Protokoll, welches einfach über Sockets angewendet werden kann.

14.5 Serialisierung von Objekten

Damit Objekte wie Meldungen von einer Zelle zur nächsten übertragen werden können, müssen Objekte serialisiert werden. Mit anderen Worten, Objekte müssen in einen Byte Stream umgewandelt und beim Kommunikationspartner wieder aus dem Byte Stream erstellt werden können. Dabei werden ausschliesslich die Attribute eines Objekts im Byte Stream gespeichert.

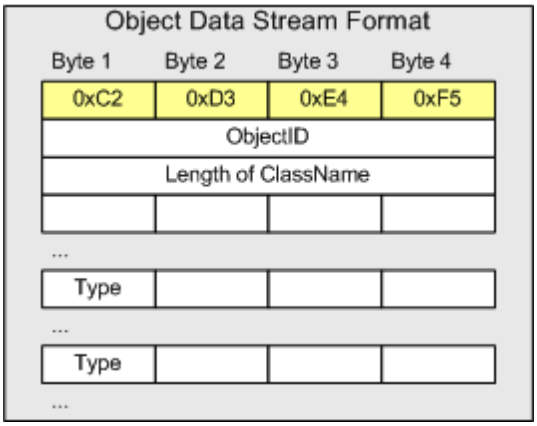


Abbildung 37: Aufbau des Serialiserungsprotokoll

Zur Serialisierung von Objekten wurde ein einfaches binäres Format definiert. Es besteht aus einem Kopfteil und aus den Daten der Objektattribute. Der Kopfteil wird durch eine 4 Byte lange Kennung initialisiert. Damit kann ein Stream als Objekt-Stream identifiziert werden. Weitere 4 Bytes dienen der Objekt ID. Am Schluss des Kopfteils wird die Länge des Klassennamens und der Klassennamen selbst angegeben. Dieser Namen dient der Objektfabrik die richtige Klasse zu instanzieren.

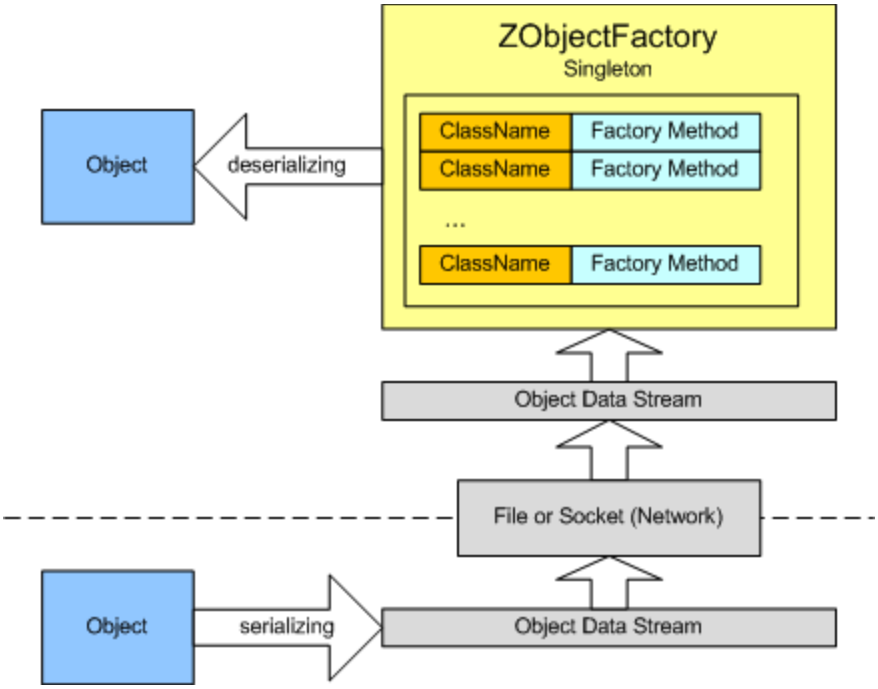


Abbildung 38: Serialisieren und Deserialisieren von Objekten mit dem Zeus-Framework

Alle serialisierbaren Objekte müssen bei der Objektfabrik registriert sein, damit die richtige Klasse instanziert werden kann. Die Registrierung erfolgt beim Programmstart, kann aber auch während der Programmlaufzeit geschehen. Sie muss aber zwingend vor dem Eintreffen des Byte Streams erfolgen.

Das Zeus-Framework unterstützt folgende Attributtypen:

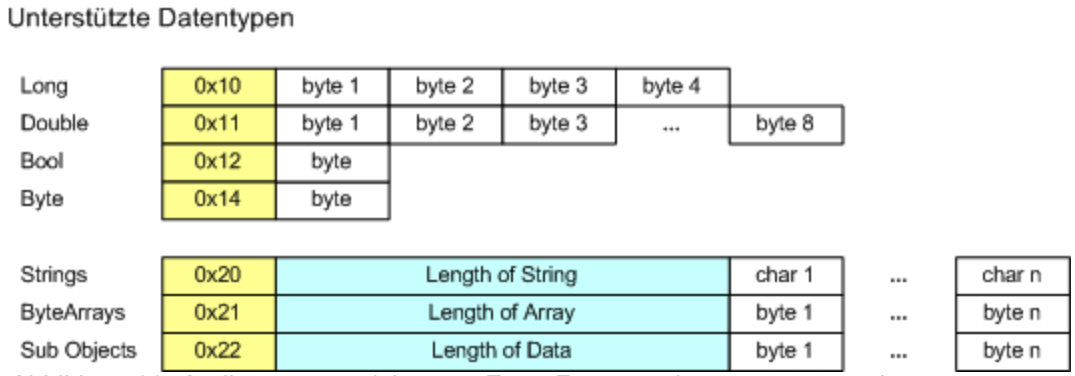


Abbildung 39: Attributtypen welche vom Zeus-Framework unterstützt werden.

Die Entwicklungsumgebung des Zeus-Frameworks stellt verschiedene Makros zur Verfügung, um möglichst einfach serialisierbare Objekte zu implementieren.

Folgendes Makro dient dem Registrieren der Klasse:

Makro zum Registrieren

```
{
  ...
  OBJECTFACTORY_REGISTER_CLASS(TBinaryMessage);
  ...
}
```

Tabelle 9: Registrieren von serialisierbaren Objekten, hier der Klasse `TBinaryMessage`.

Für die Programmierung von serialisierbaren Objekten stehen folgende Makros zur Verfügung:

Serialisierbares Objekt

```
class TSerialTestObject : public TZObject, public ISerializable
{
  public:
    TSerialTestObject();
...
    //Serializing macros
    SERIAL_START(TSerialTestObject, L"TSerialTestObject");
      SERIAL_LONG(m_lValue)
      SERIAL_DOUBLE(m_dValue)
      SERIAL_STRING(m_strValue)
      SERIAL_OBJECT(TSerialTestSubObject, m_pObject)
    SERIAL_END
...
  private:
    ///Long value
    long m_lValue;
    ///Double value
    double m_dValue;
    ///String value
    TString m_strValue;
    ///An other serializable Object
    TSerialTestSubObject* m_pObject;
};

//Implementationsteil *.cpp
SERIAL_CONSTRUCTOR(TSerialTestObject);
```

Tabelle 10: Implementation eines serialisierbaren Objekts

Folgende Klassen dienen zum Serialisieren bzw Deserialisieren von Objekten.

API	
`ISerializable`	Schnittstelle für die serialisierbaren Objekte.
`TZObjectFactory`	Fabrik zum Erstellen von Objekten aus einem Byte Stream.

➔ Extension:

Die Serialisierung kann durch weitere Datentypen erweitert werden. Die Datei `zeusbase/System/Interfaces/ISerializableImplHelper.hpp` beinhaltet für jeden Datentypen Makros. Das folgende Beispiel zeigt wie eine Liste von `double`-Werten serialisiert werden könnte (als Datentyp):

ID der Liste:

```
#define SERIAL_DOUBLELIST_ID  0x30
```

Makro zum Prüfen

```
#define CHECK_DOUBLELIST(stream) CHECK_ATTRIBUTE(stream,
SERIAL_DOUBLELIST_ID)
```

Serialisierung und Deserialisierung von Listen mit `double`-Werten

```
#define SERIAL_DOUBLELIST(attrib) \
if (ulRetval == RET_NOERROR) \
{ \
  if (lMode == 2) \
  { \
    stream_out->writeByte(SERIAL_DOUBLELIST_ID); \
    long lAttCount = obj_old->attrib.getCount(); \
    stream_out->writeLong(lAttCount); \
    for(int i = 0; i < lAttCount; i++) \
    { \
      stream_out->writeDouble(obj_old->attrib.getItemCopy(i)); \
    } \
  } \
  else if (CHECK_DOUBLELIST(stream_in)) \
  { \
    long lAttCount = stream_in->readLong(); \
    for(int i = 0; i < lAttCount; i++) \
    { \
      obj_new->attrib.add(stream_in->readDouble()); \
    } \
  } \
  else \
  { \
    ulRetval = RET_REQUEST_FAILED;\
  } \
}
```

14.5.1 Streaming

Mit den Streaming-Klassen können Daten aus Dateien und Puffer einfach gelesen und geschrieben werden.

API	
`IInputStream`	Schnittstelle fürs Lesen aus Streams.
`IOutputStream`	Schnittstelle fürs Schreiben in Streams.
`TByteArrayInputStream`	Streaming-Klasse zum Lesen aus einem Puffer.
`TByteArrayOutputStream`	Streaming-Klasse zum Schreiben in einem Puffer.
`TFileInputStream`	Streaming-Klasse zum Lesen aus einer Datei.
`TFileOutputStream`	Streaming-Klasse zum Schreiben in eine Datei.

14.6 Remote Method Invocation mit C++

Durch eine Implementation von Remote Method Invocation (RMI) kann das Zeus-Framework für verteilte Applikationen eingesetzt werden. Folgende Klassen und Objekte sind von Interesse:

API	
`Naming`	Objekt des Naming Service. Durch dieses Objekt können Remote Objects nachgeschlagen werden, oder es können eigene Remote Objects registriert werden.
`IRemoteObject`	Alle Remote Objects müssen diese Schnittstelle implementieren. Die Hilfsklasse `TAbstractRemoteObject` implementiert die wichtigsten Methoden.
`TAbstractSkel`	Hilfsklasse für die Implementation des Skeleton.

API	
`TAbstractStub`	Hilfsklasse für die Implementation der Stub-Klasse.
`ISerializable`	Schnittstelle für die serialisierbaren Objekte.

Damit das RMI funktioniert, muss auf einem Rechner ein Namensdienst gestartet werden. Die Applikation muss dann den Naming Service (lokal) mit dem Namensdienst verbinden. Weitere Informationen zum RMI für C++ sind auf dem Web unter www.xatlantis.ch/examples/rmi_example.html zu finden.

14.7 Klassen für künstliche Intelligenz

Das Zeus-Framework bietet einige Klassen für Fuzzy Logik und genetische Algorithmen an:

API	
`TFuzzySet`	Das Fuzzy Set beinhaltet eine Menge zum Rechnen mit Fuzzy Logik.
`TFuzzyLogic`	Fuzzy Logik Klasse.
`IGAGene`	Interface für Gene (Genstrukturen)
`TGAChromosone`	Sammlung von Genen. Implementiert das Kreuzen von Gen-Strängen.
`TGAIndividual`	Ein Individuum.
`TGAPopulation`	Sammlung von Individuen. Durch das Bilden von Generationen werden immer bessere Individuen gezüchtet.

Weitere Informationen zu genetischen Algorithmen sind auf dem Web unter www.xatlantis.ch/examples/ga_example.html zu finden.

15 Definition von X-Objekten in XML

Dieser Abschnitt befasst sich mit der statischen Strukturdefinition von X-Objekten in einem Objektbaum. Dieser Objektbaum ist in XML abgebildet, da sich diese Beschreibungssprache optimal für solche Strukturen eignet.

15.1 Allgemeine XML Spezifikation

Die Definition in XML ist sehr offen spezifiziert. Ein XML Knoten, welcher ein X-Objekt repräsentiert, kennzeichnet sich lediglich durch folgende Attribute:

- Knotennamen: Gibt dem Objekt einen Namen.
- Attribut `ClassName`: Name der Klasse von welcher das Objekt erzeugt werden soll.
- Attribut `CodeModule`: Name des Code-Moduls, in welchem die Klasse implementiert ist.

➔ Es können beliebige weitere Attribute definiert werden, welche Daten für die konkreten Objekte enthalten. Der XML Knoten kann auch Kinderknoten haben, die keine X-Objekte repräsentieren.

Der Wurzelknoten des XML Knotens muss immer ein Objekt erzeugen, welches die Schnittstelle `IXRootObject` implementiert.

15.1.1 Objekte

Die Spezifikation schreibt nicht vor, dass sich Objekte im XML eindeutig von einander unterschieden. Je nach Anwendung ist es aber erwünscht, dass jedes Objekt eindeutig identifizierbar ist. Das XML Objekt wird durch den Namen des Objekts (Knotennamen) und dem Attribute ID zum Unikat.

Objekt: `<Name><ID>`

Beispiel von XML Objekt Tree

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<RootObject ID="1" ClassName="TXRootObject">
  <SubObject1 ID="1" ClassName="TXObject1"/>
  <SubObject1 ID="2" ClassName="TXObject1"/>
  <SubObject2 ID="1" ClassName="TXObject2"/>
</RootObject>
```

Listing 1: Beispiel einer XML Objekt Konfiguration mit eindeutig identifizierbaren Objekten.

15.1.2 Attribute

Das Objekt kann Attribute enthalten. Es gibt 3 Schemas, wie Attribute in das XML eingefügt werden können.

Das erste Schema definiert, dass Attribute des Objekts auch als Attribute des XML Knotens eingefügt werden. Dabei können alle primitiven Werte einfach und schnell abgebildet werden. Der Vorteil dieser Art ist, dass sie weniger Speicher verwendet und der Zugriff schnell ist.

Erstes Schema

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<RootObject ID="1" ClassName="TXRootObject" Label="Begruessung"
  Text="Hallo Welt!">
  <SubObject1 ID="1" ClassName="TXObject1" Status="OK"/>
  <SubObject1 ID="2" ClassName="TXObject1" Status="NA"/>
  <SubObject2 ID="1" ClassName="TXObject2"/>
</RootObject>
```

Listing 2: Beispiel einer XML Objekt Konfiguration mit Attributen, die auch als Attribute im XML Objekt eingefügt wurden.

Das zweite Schema definiert, dass die Attribute als Unterknoten `<Attribute>` eingefügt werden. Der Name und der Wert des Attributs werden direkt in diesem Knoten definiert, wie zum Bsp. `<Attribute Name="Name" Value="MyName">`. Dies hat den Vorteil, dass auch Listen einfach angegeben werden können. Es braucht aber schon erheblich mehr Speicher grosse Objektbäume abzubilden.

Zweites Schema

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<RootObject ID="1" ClassName="TXRootObject" AttributeSchema="2">
  <Attribute Name="Label" Value="Begruessung"/>
  <Attribute Name="Text" Value="Hallo Welt!"/>
  <SubObject1 ID="1" ClassName="TXObject1">
    <Attribute Name="Status" Value="OK"/>
  </SubObject1>
  <SubObject1 ID="2" ClassName="TXObject1"/>
    <Attribute Name="Status" Value="NA"/>
  </SubObject1>
  <SubObject2 ID="1" ClassName="TXObject2"/>
</RootObject>
```

Listing 3: Beispiel einer XML Objekt Konfiguration mit Attributen, die als Unterknoten zum XML Objekt eingefügt wurden.

Das dritte und flexibelste Schema definiert, dass die Attribute als Unterknoten eingefügt, die Attributwerte im Gegensatz zur 2. Variante aber als Textknoten eingefügt werden. Dies hat den grossen Vorteil, dass auch komplexe Attributwerte, wie zum Bsp. ein Textdokument gespeichert werden kann.

Drittes Schema

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<RootObject ID="1" ClassName="TXRootObject" AttributeSchema="3">
  <Attribute Name="Label">Begruessung</Attribute>
  <Attribute Name="Text">Hallo Welt!</Attribute>
  <SubObject1 ID="1" ClassName="TXObject1">
    <Attribute Name="Status">OK</Attribute>
  </SubObject1>
  <SubObject1 ID="2" ClassName="TXObject1"/>
    <Attribute Name="Status">NA</Attribute>
  </SubObject1>
  <SubObject2 ID="1" ClassName="TXObject2"/>
</RootObject>
```

Listing 4: Beispiel einer XML Objekt Konfiguration mit Attributen, die als Unterknoten zum XML Objekt eingefügt wurden. Die Attributwerte sind als Textknoten angegeben, was eine grosse Flexibilität zulässt.

Diese 3 Schemas dürfen für ein XML Objekt nicht gemischt werden. Welche Art für ein XML Objekt verwendet wird, wird mit dem Attribut `AttributeSchema="SchemaNr"` angegeben. Ist für ein Vaterknoten bereits ein Schema festgelegt, wird dieses von den Kinderknoten geerbt. standardmässig ist das Schema 1 gesetzt.

15.2 Erweiterte Definition zum ClassLoader

Der XML Objektbaum kann Objekte besitzen, die zur Laufzeit noch nicht bekannt sind. Dabei handelt es sich um so genannte externe Objekte, welche in einem Code-Modul implementiert sind. Diese Code-Module müssen während dem Aufbauen des Baums geladen werden. Beim XML Objekt wird das Attribut `CodeModule="path/filename"` angegeben. Das Attribut setzt sich aus einem relativen Pfad und dem Namen einer Bibliothek (ohne Endung) zusammen. Der Klassenlader (ClassLoader) fügt automatisch die korrekte Endung an (unter Linux mit Endung *.so, unter Windows mit Endung *.dll). Der Basispfad aller Module ist standardmässig auf `./modules` gesetzt.

Weiter können externe XML Referenzen angegeben werden. Diese werden während dem Erstellen der Objekthierarchie geladen. Darin können sich weitere Teilbäume der Objekthierarchie befinden. Die XML Referenzen müssen als Dateinamen mit relativem Pfad angegeben werden. Der Basispfad ist auch hier auf `./modules` gesetzt.: Zum Beispiel: `XMLRef="path/filename.xml"`. Diese Angabe kann nicht bei allen Objekttypen gemacht werden. Das Objekt muss die Schnittstelle `IXLoaderObject` implementieren.

Bei den ladbaren Objekten (`IXLoaderObject`) kann das Attribut `CreateChildren="x"` gesetzt werden, um die Objekthierarchie gestaffelt aufzubauen. Dieses Attribut unterliegt dem Attributschema. Mit dem Wert `x=1` werden alle Kinderobjekte sofort erstellt. Beim Wert `x=0` werden die Kinderobjekte erst erstellt, wenn sie jemand braucht.

Durch das Einsetzen der drei Erweiterungen, können Teilbäume ausgelagert und erst bei Bedarf geladen werden. Zudem können Objekte in Code-Modulen implementiert werden.

Die 3 Erweiterungen

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<RootObject ID="1" ClassName="TXRootObject" AttributeSchema="3">
  <ModuleA ID="1" ClassName="TModule" CodeModule="ModuleA">
    <Attribute Name="CreateChildren">1</Attribute>
    <ModuleALogic ClassName="TLogicA">
  </ModuleA>
  <ModuleB ID="1" ClassName="TModule" CodeModule="test/ModuleB"
    XMLRef="test/ModuleB.xml"/>
    <Attribute Name="CreateChildren">0</Attribute>
  </ModuleB>
</RootObject>
```

Listing 5: Beispiel welches die 3 Erweiterungen nutzt. Zu beachten ist, dass hier das Attributschema 3 verwendet wird.

16 Entwickeln von Code-Modulen

Damit Bibliotheken (Code-Module) im Zeus-Framework gebraucht werden können, müssen zwei Funktionen exportiert werden.

API	`MODULEXPORT_PRE unsigned long MODULEXPORT_POST registerLibrary(NAMESPACE_Zeus::ISingletonManager* manager)`
	Diese Funktion wird beim Laden der Bibliothek aufgerufen. Die Bibliothek kann durch den Singleton-Manager alle System-Singletons ermitteln und seine statischen Singletonreferenzen auf die System-Singletons setzen.

API	`MODULEXPORT_PRE unsigned long MODULEXPORT_POST unregisterLibrary(NAMESPACE_Zeus::ISingletonManager* )\`
	Diese Funktion wird kurz vor dem Entladen der Bibliothek aufgerufen.

Bei den meisten Bibliotheken wird der Code zur Registrierung immer gleich aussehen. Deshalb kann hier das Makro verwendet werden. Es initialisiert die wichtigsten Singletons und erzeugt einen Logger zur Ausgabe von Informationen (in eine Datei oder in eine Konsole).

Initialisieren der Bibliothek

```
MregisterLibrary(L"MeineBibliothek");
MunregisterLibrary();
```

Es gibt 2 Arten wie ein Code-Modul gebraucht werden kann:

- Das Code-Modul wird nur zur Erzeugung von Objekten gebraucht
- Das Code-Modul wird durch ein X-Objekt vom Typ `TModule` geladen und eine Session wird erstellt.

Beide Varianten sind auch in Kombination einsetzbar. Am häufigsten wird jedoch die erste Variante gewählt. Hier eine kurze Erläuterung der Funktionalität und der Unterschiede

16.1 Code Modul zur Erzeugung von Objekten

Die Bibliothek `xy` wird geladen, wenn im Objektbaum ein Objekt erzeugt werden soll, welches das Attribut `CodeModule="xy"` gesetzt hat. Danach wird versucht das Objekt durch eine exportierte Funktion der Bibliothek zu erzeugen. Das Objekt wird direkt zurückgegeben und kann nun in den Objektbaum eingehängt werden.

Die exportierten Funktionen heissen immer **`create`**`TMyClass`, wobei `TMyClass` für einen Klassennamen steht. Der gleiche Klassennamen muss im XML im Attribut `ClassName` angegeben werden.

Um eigene X-Objekt Klassen einfach zu exportieren gibt es im C-Header `zeusdev/AbstractModuleSession.h` Makros.

Fabrikfunktion der Bibliothek

```
#include <InputCell.h>
#include <zeusdev/AbstractModuleSession.h>

USING_NAMESPACE_Zeus
MExportXObjectFactory(TInputCell);
```

Tabelle 11: Beispiel zeigt das Exportieren einer Fabrikfunktion zum Erzeugen des X-Objekts `TInputCell`

16.2 Code Modul zum Erstellen einer Session

Ladet ein X-Objekt vom Typ `TModul` eine externe Ressource in Form einer Bibliothek, wird eine Modulsession erzeugt. Die Session dient fortan zur Kommunikation zwischen dem X-Objekt und der Bibliothek. In einer Session können beliebige Objekte von der Bibliothek erzeugt werden. Diese Möglichkeit dient all jenen Bibliotheken die keine X-Objekte implementieren wollen.

Zur bidirektionalen Kommunikation zwischen der Bibliothek und dem Modul werden 2 Schnittstellen ausgetauscht:

API	
`IModuleSession`	Durch diese Schnittstelle kann auf die Session einer Bibliothek zugegriffen werden.
`IZeusAPI`	Schnittstelle für den Zugriff auf den X-Objektbaum
`TAbstractModuleSession`	Diese Klasse ist eine Hilfsimplementierung um einfach und schnell eigene Modulsessionen zu entwickeln.

Die Schnittstelle der Ressource wird `IModuleSession` genannt. Durch diese kann das Framework mit der gewünschten Bibliothek kommunizieren und Objekte erzeugen, bzw. anfordern.

Die Schnittstelle des Frameworks ist das `IZeusAPI`. Durch diese Schnittstelle kann die Bibliothek auf das Framework oder auf eine andere Bibliothek des Frameworks zugreifen.

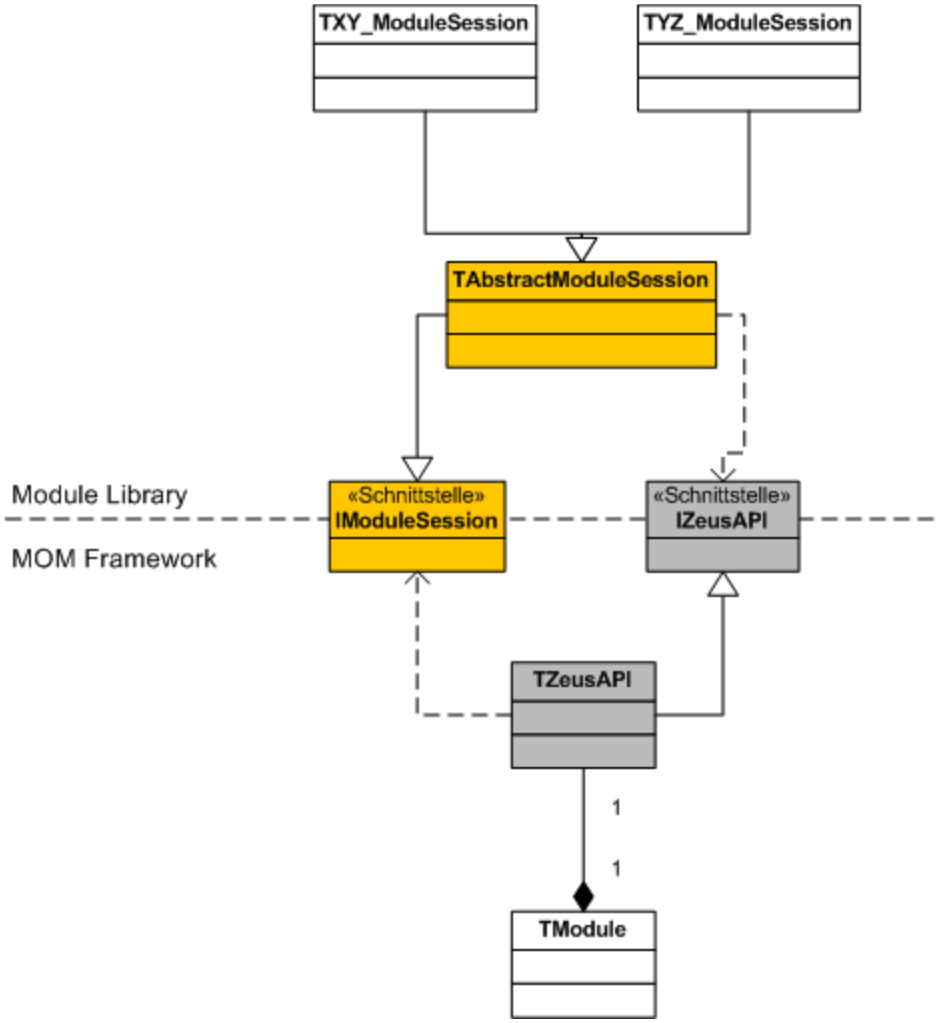


Abbildung 40: Schnittstellen, welche beim Laden der Bibliothek ausgetauscht werden.

Wenn mehrere X-Objekte vom Typ `TModul` die gleiche Bibliothek laden möchten, wird pro X-Objekt eine neue Session erzeugt. Die Bibliothek wird aber nur einmal geladen.

➔ Werden die Bibliotheken erst bei Gebrauch geladen, kann einerseits Speicher und anderseits Zeit beim Hochfahren eingespart werden. Das Attribut im entsprechenden XML Knoten muss mit `CreateChildren="0"` eingestellt werden

Um die Modulsession oder eigene Klassen einfach zu exportieren gibt es im C-Header `zeusdev/AbstractModuleSession.h` Makros.

Fabrikfunktion der Bibliothek

```
#include <TestModuleSession.h>

USING_NAMESPACE_Zeus
MExportObjectFactory(IModuleSession, TTestModuleSession);
```

Tabelle 12: Beispiel zeigt das Exportieren einer Fabrikfunktion zum Erzeugen von normalen Objekten, wie zum Beispiel eine Modulsession

16.3 Beispiel einer Code-Modul Implementation

Im folgenden Beispiel wird gezeigt, wie ein Code-Modul zu entwickeln ist. Das Code-Modul beinhaltet ein X-Objekt, welches exportiert wird.

Das Beispiel setzt sich aus 3 Dateien zusammen:

- `ExampleModule.cpp` : Exportierte Funktionen der Bibliothek
- `XObjectExample.h`: Definition des X-Objekts
- `XObjectExample.cpp`: Implementation des X-Objekts

Alle exportieren Funktionen

```
#include <stdio.h>
#include <XObjectExample.h>
#include <zeusbase/System/XObjectFactory.h>
#include <zeusbase/System/LibraryManager.h>
#include <zeusdev/AbstractModuleSession.h>

USING_NAMESPACE_Zeus

USING_NAMESPACE_Zeus

////////////////////////////////////////////
// Exported Factories
MExportXObjectFactory(TXObjectExample );
////////////////////////////////////////////

MregisterLibrary(L"ExampleModule");
MunregisterLibrary();
```

Tabelle 13: Dieser Code steht in der CPP Datei ExampleModule.cpp

Das `TXObjectExample` ist ein normales X-Objekt. Welche Möglichkeiten und welche Makros noch zur Verfügung stehen um X-Objekte zu erstellen, ist im Kapitel [] auf Seite beschrieben.

Definition des X-Objekts

```
class TXObjectExample : public TXObject
{
  public:
    TXObjectExample(IXMLNode* node);

    //Methods of IXObject
    virtual bool MQUALIFIER freeze();
    virtual bool MQUALIFIER unfreeze();

    REG_SUB_BEGIN(TXObjectExample)
    REG_SUB_ADD(TXObjectExample, L"TXObjectExample");
    REG_SUB_END
...
  protected:
    virtual ~TXObjectExample();
...
};
```

Tabelle 14: Definition des X-Objekts. Die Makros dienen der Registrierung bei der Objektfabrik.

Implementation des X-Objekts

```
TXObjectExample::TXObjectExample(IXMLNode* node) : TXObject(node)
{...}

TXObjectExample::~TXObjectExample()
{...}

bool MQUALIFIER TXObjectExample::freeze()
{
  bool bRetval = TXObject::freeze();
  if (bRetval) {...}
  return bRetval;
}

bool MQUALIFIER TXObjectExample::unfreeze()
{
  bool bRetval = TXObject::unfreeze();
  if (bRetval) {...}
  return bRetval;
}
```

Tabelle 15: Implementation des X-Objekts

Die Methoden `freeze()` und `unfreeze()` sind im Kapitel [] Seite erklärt.

17 Das Sicherheitskonzept

In diesem Kapitel wird das Sicherheitskonzept des Zeus-Frameworks erläutert. Nicht umgesetzte Mechanismen oder Aspekte sind klar deklariert.

Das Sicherheitskonzept umfasst das Laden von den Code-Modulen und der Überwachung von laufenden Zellen.

17.1 Sicheres Laden von Code-Modulen

Das Konzept des sicheren Ladens von Code-Modulen wurde in der aktuellen Version des Zeus-Frameworks umgesetzt. Die Eigenschaft kann durch die Einstellung `zeus.LibraryManager.Security` in der Konfiguration ein oder ausgeschaltet werden.

Ob ein Code-Module geladen werden kann oder nicht, liegt an seiner Signatur. Diese Signatur wird vor jedem Laden erstellt und mit einem Eintrag aus einer Datenbank verglichen.

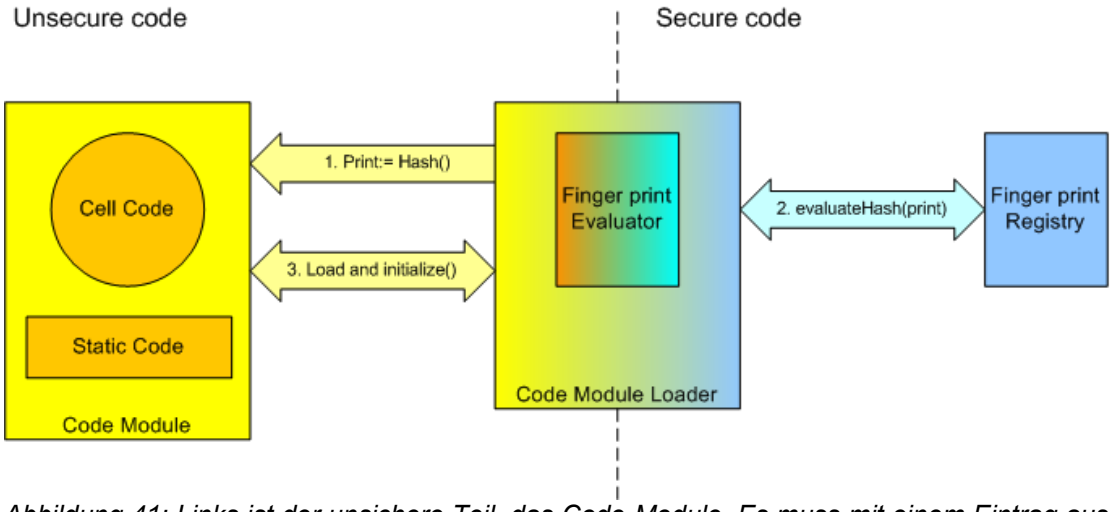


Abbildung 41: Links ist der unsichere Teil, das Code-Module. Es muss mit einem Eintrag aus der Datenbank verglichen werden, bevor es geladen wird.

Das aktuelle Zeus-Framework generiert den Fingerabdruck mit dem Whirlpool

Algorithmus. Die Einträge in der Datenbank (Datei `./bin/security.db`) sind noch unverschlüsselt.

➔ Extension:

Die Datenbank könnte mit einem Krytologie Algorithmus verschlüsselt werden.

Die Registrierung von Code-Modulen ist im Anhang beschrieben (Benutzerhandbuch).

17.2 Überwachung der Zellen

Zur Überwachung der Zellen existiert in der aktuellen Version des Zeus-Frameworks nur die Klasse `TCellSurvey`, welche die Zelle auf ihre Auslastung überprüft und den Klon-Algorithmus steuert (Siehe Klonen). Es existiert noch keine Komponente, die das Laufzeitverhalten aus dem sicherheitstechnischen Aspekt betrachtet und schädliche Zellen erkennt bzw. eliminiert. Diese Überwachung ist ein Teil der Entwicklung von mobilen Zellen, welche in der Spezifikation des CCM beschrieben sind.

➔ Extension:

Entwicklung von mobilen Zellen und Erweitern des Sicherheitsdiensts. Die Zellen müssen auch während der Laufzeit überwacht werden. Überschreitet eine Zelle ihre Kompetenzen, muss sie entfernt werden (Thread kill).

Diplomarbeit

Projekthandbuch

18 Einleitung Projekthandbuch

Das Projekthandbuch dient als einheitliche Handlungsgrundlage für alle Projektbeteiligten und legt damit den allgemein gültigen, technischen und organisatorischen Rahmen fest.

Dieses Dokument ist die Weiterführung des bereits bestehenden Projekthandbuchs der Projektarbeit und beschreibt im wesentlichen den Ablauf der Diplomarbeit. Die in der Projektarbeit abgeschlossenen Ziele und Arbeitspakete werden hier nicht mehr erwähnt.

19 Projektbeschreibung

Das Projekt Cell Computing Model (CCM) befasst sich mit einem neuen Paradigma der Informatikwelt. Das Paradigma versucht ein Software Modell zu definieren, welches Aufbau und Arbeitsweise von biologischen Organismen zum Vorbild nimmt. Ähnliche Denkansätze liegen etwa dem *Grid Computing* oder den *evolutionären Algorithmen* zu Grunde.

Die Diplomarbeit hat ein Software Framework zum Ziel, welches - nach biologischem Vorbild - auf dem Zusammenwirken von miteinander kommunizierenden dynamisch erzeugten Software-Zellen beruht.

Das Projekt wird als Open Source Projekt veröffentlicht. Somit können keine Patentrechte seitens Fachhochschule oder des Diplomanden geltend gemacht werden. Das Projekt ist öffentlich unter www.xatlantis.ch zugänglich.

19.1 Ausgangssituation

In der Projektarbeit wurde eine Grundlage in Form eines Frameworks erschaffen. Die Diplomarbeit führt diese Arbeit nun weiter, indem weitere Funktionen und Verhaltensweisen umgesetzt werden. Dazu wurde das Projekt an der HTI Bern als Diplomarbeit beantragt und genehmigt.

Das Projekt ist eine Grundlagenarbeit zu dem Thema und kann im weitesten Sinne als Forschungsarbeit beschrieben werden.

Aufbauend auf der Projektarbeit sollen insbesondere folgende Aufgaben angegangen werden:

- Erarbeiten einer Spezifikation zum Cell Computing Model
- Sicherheitskonzept. Erkennen der Zellen mit *finger print* Technik
- Beispiel-Implementation eines Bildverarbeitungsprozesses mit dem erstellten Framework
- Erweiterung des Systems mit mobilen Zellen, welche sich autonom im Netz verteilen können (fakultativ)

19.1.1 Bestehende Produkte zu CCM

Zu dem Begriff CCM wurde bereits eine Spezifikation mit Analyse geschrieben. Die genaue Definition des Begriffs wird im Dokument [CCM1]erläutert.

Weiter wurde ein Framework erschaffen, das Zeus-Framework, welches als Grundlage zur Entwicklung von CCM-Applikationen dient.

Folgende Funktionen sind bereits implementiert:

- dynamische, modulare Software Struktur
- Klonen und Abbauen von Zellen
- Kommunikation zwischen Zellen über Systemgrenzen hinweg

Eingerichtet ist auch eine Internet-Plattform unter http://www.xatlantis.ch.

19.2 Problemstellung

Die Aufgabenstellung der HTA Bern wurde im Dokument [DIP-ISWP1]verfasst. Die Aufgabenstellung/Problemstellung soll hier noch etwas genauer erläutert werden:

19.2.1 Spezifikation zum Cell Computing Model

Die Spezifikation zum Cell Computing Model muss fortlaufend ergänzt und ggf. korrigiert werden. Es fehlen noch folgende Angaben:

- Sicherheitskonzept: Wie können fremde Zellen erkannt werden und wie können solche Zellen eliminiert werden?
- Autonome Zellen, welche von einem System zum nächsten gelangen. Wie findet die Übertragung statt und nach welchen Kriterien wandert eine solche Zelle?

19.2.2 Implementation der Finger Print Technik

Umsetzen des Sicherheitskonzept, welches in der Spezifikation theoretisch beschrieben wurde. Verwenden geeigneter Algorithmen zur Erzeugung von Finger Prints und zur Verschlüsselung von sicherheitsrelevanten Daten.

19.2.3 Beispiel Implementation

Als Beispiel soll ein Verfahren aus der Bildverarbeitung implementiert werden. Dieses Beispiel soll auf dem bereits erstellten Zeus-Framework laufen und auf mehrere Rechner verteilt werden können. Es soll folgendes Verfahren angewandt werden:

- Motion Detection in einem Film (stationäre Kamera)

Die Beispiel-Implementation erfordert eine graphische Schnittstelle zum Bedienen der Applikation und zur Überwachung bzw. Einstellen der Zellen. Die Graphik wird mit QT von Trolltech implementiert [QTDOC]. Dabei muss noch geklärt werden, inwiefern QT für dieses Projekt als Open Source verwendet werden kann.

19.2.4 Reflektieren und Verbessern des System

Zusätzlich sind zu Untersuchen, ggf. zu Verbessern:

- Welche Arbeitsschritte müssen gemacht werden, oder bleiben erspart?
- Ist es einfach eine CCM Anwendung zu Programmieren?
  - Was sind die Voraussetzung, um mit dem Zeus-Framework zu programmieren (Ing. oder Lehrling)?
  - Können die Arbeitsschritte nachvollzogen werden?
- Wie schnell ist man bei der Entwicklung von Software mit dem Zeus-Framework?
  - Muss nur das wesentliche geschrieben werden?
  - Ist das Verhalten des Systems transparent?
- Ist das Verfahren performant?

19.2.5 Mobile Zellen (fakultativ)

Als Zusatz kann das System mit so genannten mobilen Zellen (Agenten) erweitert werden. Dabei wird das theoretisch erarbeitete umgesetzt. Speziell zu beachten sind folgende Aspekte:

- Wie verbreiten sich die mobilen Zellen?
  - Welches Protokoll soll verwendet werden?
  - Wie wird der Zustand der Zellen erhalten (Streaming)?
- Anwendung des Sicherheitskonzepts

19.3 Projektrisiken

Das Risiko ist mittel bis hoch einzuschätzen, dass die technischen Ziele des Projekts nicht erreicht werden, da das Projektteam nur aus einer Person besteht. Durch Krankheit, Unfall oder Ausfall durch berufliche Tätigkeit kann das Resultat des Projekts negativ beeinflusst werden.

Beim Eintreten eines der genannten Szenarien, muss mit dem Fachdozenten die Ziele neu definiert und nach möglichen Alternativen gesucht werden.

Technische Risiken bezüglich Machbarkeit gibt es keine, da auch die Erkenntnis, dass CCM nicht umsetzbar ist, ein Ergebnis dieser Diplomarbeit sein kann.

Weitere Projektrisiken sind in der folgenden Tabelle zusammengefasst:

Prio.	*Fall*	*Schadens-Verhinderung*	*Schadens-Minimierung*	*Auslöser*	*Ver-ant.*
H	Die Implementierung liefert nicht die gewünschten Resultate.	Prototyp erstellen und testen.	Durch Risikoanalyse und Testimplementationen.		Team
H	Datenhaltung: Verlust von Daten.	Backup auf verschiedene Rechner ablegen.	Backup auf verschiedene Rechner ablegen, ev periodisch CD brennen.	HD-Crash,	Team
M	Durch Krankheit von Projektmitglieder		Jeder ist informiert über den aktuellen	Unfall/Krankheit	Team

Prio.	*Fall*	*Schadens-Verhinderung*	*Schadens-Minimierung*	*Auslöser*	*Ver-ant.*
	verzögert sich das Projekt.		Projektstand.		
H	Zusätzliche Belastungen durch Arbeit in der Firma verzögert sich das Projekt.		Sorgfältige Terminpläne und vernünftiges Zeitmanagement.	Arbeitgeber	Team
N	Änderung der Projektanforderungen durch neue Erkenntnisse.		Sorgfältige Analyse zu Beginn des Projekts, Prototypen entwickeln.	Projektteam, Betreuer	Team

19.4 Ziele

Das Ziel der Diplomarbeit ist das Vertiefen der Grundlagen. Mit dem bereits entwickelten Framework Prototypen sollen die Grundlagen des CCM verifiziert, getestet und ausgewertet werden.

Das Hauptziel in der Diplomarbeit ist das Erstellen einer Beispiel-Implementation zur Bildverarbeitung.

Ziel-Nr.	Beschreibung
1	Erarbeiten einer Spezifikation zum Cell Computing Model.
2	Sicherheitskonzept. Erkennen der Zellen mit *finger print* Technik.
3	Beispiel-Implementation eines Bildverarbeitungsprozesses mit dem erstellten Framework.
4	Erweiterung des Systems mit mobilen Zellen, welche sich autonom im Netz verteilen können (fakultativ).

Tabelle 16: Ziele der Diplomarbeit 2005

19.5 Vorgehensstrategie

Um die oben genannten Ziele zu erreichen, wird ein iteratives Modell verwendet. Dabei werden periodisch Prototypen entwickelt, welche als Grundlage für weitere Entscheidungen dienen. Eine Iteration dauert ca. 1 Woche.

Fortlaufend wird die Spezifikation angepasst und begutachtet.

Projektverzögerungen werden mit Funktionsreduzierung entgegengewirkt, bis das Minimum an Muss-Funktionen erreicht wird. Deshalb sind alle Funktionen und Aufgaben priorisiert.

20 Projektspezifisches Vorgehensmodell

Als Vorgehensmodell wird ein iteratives Modell gewählt. Dabei wird das Projekt in Arbeitspakete unterteilt. Die Arbeitspakete werden sequentiell als Prototypen in das Framework integriert. Dabei spielen die Design-Reviews und Code-Reviews mit dem technischen Betreuer einerseits eine qualitätssichernde Rolle, andererseits dienen sie als Entscheidungspunkte für weiteres Vorgehen. Diese Reviews werden kontinuierlich nach jedem Designschritt und bei jeder Integration der Teilprodukte gemacht. Empfohlen wird eine Zykluszeit von 4-5 Wochen.

Da das Projektteam aus nur einem Projektsachbearbeiter besteht, und das Projekt eine Art Forschungsprojekt darstellt, wird kein Vorgehensmodell nach Hermes verwendet, keine Submodelle und somit auch kein Tailoring angewendet.

Arbeitspaket	***Priorität***
Beispiel Bildverarbeitungsprozess • Analyse der Performance • Vergleich zur herkömmlichen Anwendung	**Hoch(*)**
Sicherheitskonzept für CCM • Finger print • Security oder Polizisten-Zellen	**Hoch(*)**
Spezifikation CCM vervollständigen	**Hoch**
Mobile Zellen, die sich über das Netzwerk auf andere PCs gegeben können	Mittel
Gebrauch von Petrinetz-Strukturen zur Beschreibung von Prozessen (Theoretisch erarbeiten)	Mittel

(*): Diese Arbeitspakete sind im Anhang noch näher spezifiziert und den Aufwand geschätzt.

21 Entscheidungspunkte und auszuliefernde Ergebnisse

Während der Projektphase existieren folgende Entscheidungspunkte. Die Reihenfolge ist im Kapitel 7.2 Projektplanung definiert.

Entscheidungspunkt	Ergebnisse	Zuständigkeiten	QS
Diplomarbeit Initialisierung	Auftrag	TB	Hr. Hett
	Projekthandbuch	PSB	TB
Konzept Diplomarbeit	Analyse	PSB	TB/Experte
	Modell	PSB	TB/Experte
Realisierung Diplomarbeit	Anwendung	PSB	TB/Experte
	CCM Spezifikation	PSB	TB/Experte
Abschluss Diplomarbeit	Reflektion	PSB	TB/Experte

PSB: Projektsachbearbeiter
TB: Technischer Betreuer

22 Methoden und Werkzeuge

In diesem Kapitel sind für die einzelnen Submodelle, wie Software Engineering SE und Projektmanagement PM, die Methoden und Werkzeuge definiert.

22.1 Projektmanagement PM

Aktivität	***Methode***	***Werkzeug***
Planung erstellen	Iteratives Vorgehen	Open Office
Feinplanung erstellen	Prototyping	Open Office/HTML
Projektberichte erstellen	Dokumentation nach Vorlage	Open Office
Sitzungen planen	Nach Projektplan	E-Mail HTI Bern

22.2 Software Engineering SE

Aktivität	***Methode***	***Werkzeug***
Design erstellen	Unified Modelling Language	Visio von MS
Dokumentation für Produkt erstellen	Dokumentation nach Vorlage	Open Office
Code Dokumentation	nach Konventionen (Doxygen)	Code Editor, Doxygen
Code	nach Konventionen (C++)	KWrite, KDevelop

Aktivität	*Methode*	*Werkzeug*
Konfiguration erstellen	nach Konventionen (XML)	KWrite

23 Standards und Richtlinien

Folgende Richtlinien sind zu befolgen:

Richtlinie	*Dokument*
Codierstandard für C++	
Codierstandard für XML	
Design Patterns	
Namenskonventionen	
Integrationsbestimmungen	

24 Ergänzende Projektvereinbarungen

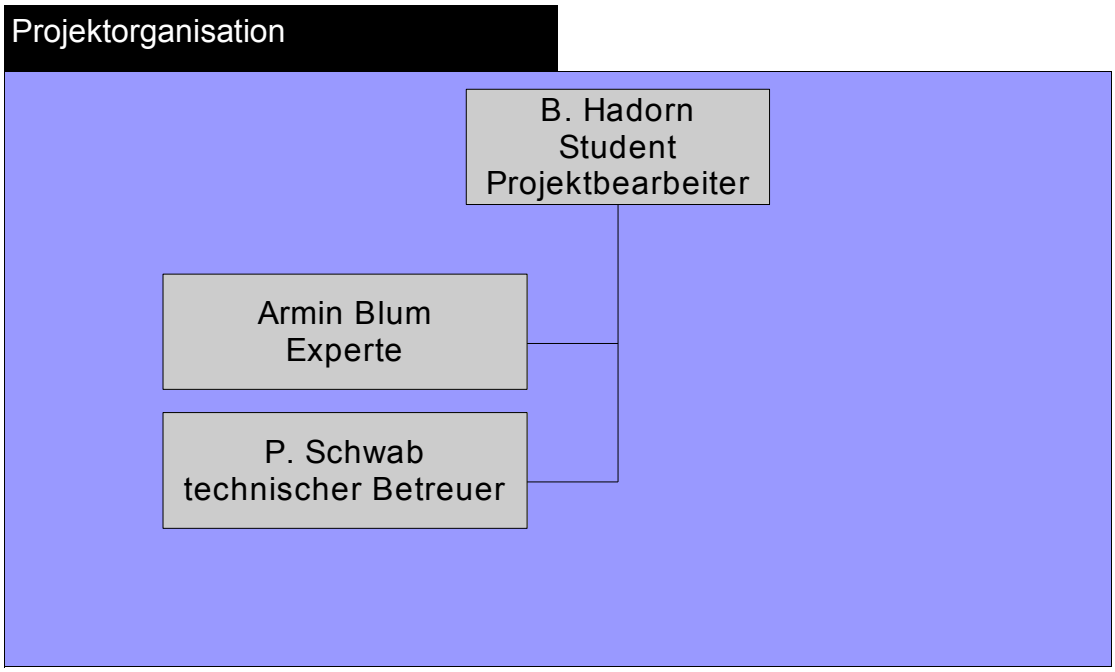
24.1 Projektorganisation

Da nur eine Person das Projekt bearbeitet, übernimmt diese die Rollen des

- Projektleiters
- Software-Architekten
- Projektsachbearbeiter
- Tester

Das Projektteam wird durch den Fachdozenten P. Schwab technisch unterstützt.

Die Diplomarbeit wird durch den Experten Armin Blum begleitet



24.2 Projektplanung

Die Reihenfolge der Arbeitspakete sind folgendermassen definiert:

1. Sicherheitskonzept für CCM
2. Beispiel Bildverarbeitungsprozess
3. Spezifikation CCM vervollständigen
4. Mobile Zellen
5. Gebrauch von Petrinetz-Strukturen

Folgende Meilensteine sind definiert:

Datum	*Zeit*	*Termin*
19.09.05	Um 0815	Start der Diplomarbeit
23.12.05		Abgabe der Diplomarbeit • Abgabe des Produkts • Abgabe der Dokumente
09.01.06	0930	Verteidigung des Projekts
20.01.06 bis 21.01.06	ab 1400	Diplomausstellung an der HTA Bern
27.01.06	ab 1700	Diplomfeier im Kursaal Bern

24.3 Aufwandschätzung

Die Aufwandschätzung wurde für die wichtigsten Arbeitspakete vorgenommen.

- Beispiel Bildverarbeitungsprozess
- Sicherheitskonzept für CCM
- Weiterentwicklung des Frameworks und Hilfsklassen

- CCM Spezifikation erstellen

Für die Diplomarbeit sind pro Woche 20 Lektionen geplant.

14 Wochen x 20 Lektionen = 280 Lektionen Diplomarbeit

Die Aufwandschätzung wird in Tagen angegeben. Dabei besteht ein Schultag aus 8 Lektionen für einen Studenten der 100% arbeitet. Für berufsbegleitende Studenten ist der Prozentsatz geringer (ca. 60-80%).

280 Lektionen / 8 Lektionen pro Tag = 35 Schultage Aufwand

Arbeitspaket	***Aufwand in Schultage***
Beispiel Bildverarbeitungsprozess	14
Sicherheitskonzept für CCM	10
Spezifikation CCM vervollständigen	3
Reviews und Sitzungen (Pauschal über alle Arbeitspakete)	2
Projektdokumente (Handbuch, Planung und Reflektion)	2
Web Seiten	3
Reserve	1
Total Schultage	35

Die 2 ersten Arbeitspaket sind in den folgenden Unterkapitel noch näher spezifiziert. Die Arbeitspakete sind in dem Projektplan in zeitliche Abhängigkeit dargestellt. Auf einen Netzplan wurde verzichtet, da die Arbeitspakete nur sequentiell abgearbeitet werden (nur ein Projektsachbearbeiter).

24.3.1 Arbeitspaket: Beispiel Bildverarbeitungsprozess

Arbeitspaket	**Beispiel Bildverarbeitungsprozess**
Priorität	Hoch
Projekt Verantwortlicher	Benjamin Hadorn
QS	Hr. P. Schwab
Projektmitarbeiter	Benjamin Hadorn

Eingaben/Ressourcen/Knowhow		***Menge***	***Aufwand (d)***
	Spezifikation Cell Computing Model	1	
	Diplomarbeit Rahmenbedingungen		
	Zeus Framework	1	1
	Grundlagen zu Bildverarbeitung und Computer Vision Motion Detection Algorithmus	1	

Aktivitäten		***Parameter***	***Menge***	***Aufwand (d)***
	Erstellen eines UML Modells der Anwendung	Diagramme	1	2
	Erstellen der Grundstruktur	Code Module	1	2
	Erstellen der graphischen Oberfläche	Code Module	1	2
	Erstellen der Zellen mit Algorithmen	Code Module	1	6
	Testen der Anwendung			1
	Code Review	Code Module	4	Pauschal

Produkte / Resultate		***Menge***	***Aufwand (d)***
	UML Modell		
	Anwendung zur Motion Detection		
Total in Tage			14

Bemerkungen
Die Code Module der Anwendung setzen sich zusammen aus: • EVA Zellen • Protokolle • GUI Module • Testklassen

24.3.2 Arbeitspaket: Sicherheitskonzept für CCM

Arbeitspaket		**Sicherheitskonzept für CCM** (Finger prints)		
Priorität		Hoch		
Projekt Verantwortlicher		Benjamin Hadorn		
QS		Hr. P. Schwab		
Projektmitarbeiter		Benjamin Hadorn		
Eingaben/Ressourcen/Knowhow			***Menge***	***Aufwand (d)***
	Bestehende Spezifikation Cell Computing Model		1	
	Diplomarbeit Rahmenbedingungen			
	Mathematische/ Kryptologische Grundkenntnisse			
	Bekannte Kryptographie-Algorithmen		2	1
Aktivitäten		***Parameter***	***Menge***	***Aufwand (d)***
	Erarbeiten des Konzepts	Modell	1	3
	Implementieren des Konzepts.	Code Module	1	4
	Testen der Verhaltensweisen	Tests	1 bis 4	2
	Code Review	Code Module	4	Pauschal
Produkte / Resultate			***Menge***	***Aufwand (d)***
	Sicherheitsklassen im Framework			
	Tests			
Total in Tage				10
Bemerkungen				
Der Algorithmus für Finger Prints wird nicht selber entwickelt, sondern ein bestehender eingesetzt.				

Diplomarbeit

Schlussbericht

25 Einleitung Schlussbericht

Im Schlussbericht der Diplomarbeit sollen die erreichten und nicht erreichten Ziele aufgezeigt werden (SOLL-IST Vergleich). Zudem ist eine Reflektion zur erarbeiteten Lösung enthalten, die aufzeigt, wo die Vorteile, bzw. Nachteile beim Einsatz des Frameworks liegen.

26 Soll-Ist Vergleich

Dieses Kapitel befasst sich mit dem Soll-Ist Vergleich. Dabei werden die Resultate, Termine und Aufwände berücksichtigt. Nicht enthalten sind die Kosten. Da es sich um ein reines Schulprojekt handelt und auf bestehenden Ressourcen zurückgegriffen wurde, war das Projekt kostenneutral.

Es wurden alle geplanten Muss-Ergebnisse erfolgreich umgesetzt. Zusätzlich wurde ein RMI (Remote Method Invocation) für C++ geschrieben, was einen Mehraufwand von 8 Schultagen bedeutete. Die Begründung des Mehraufwands ist weiter unten beschrieben.

Folgende Ergebnisse wurden fertig gestellt:

Ergebnis	***Realisiert Ja/Nein***	***Geplanter Aufwand[5]***	***Effektiver Aufwand[5]***
Beispiel Bildverarbeitungsprozess und Mandelbrot-Menge[6]	Ja	14	15
Sicherheitskonzept für CCM	Ja	10	4
Spezifikation CCM vervollständigen	Ja	3	5
Reviews und Sitzungen	Ja	2	2
Projektdokumente	Ja	2	3
Web Seiten	Ja	3	5
RMI für C++	Nicht geplant		8

5 Aufwand in Schultage mit 8 Lektionen pro Tag

6 Ergebnis wurde nicht von Anfang an in der Planung berücksichtigt.

26.1 Abweichungen zu den geplanten Termine und Ergebnissen

Das fakultative Arbeitspaket zur Umsetzung der mobilen Zellen, wurde nur theoretisch behandelt. Im Sicherheitskonzept wurden nicht alle Aspekte zur Überwachung von Zellen umgesetzt, da diese nur mit mobilen Zellen hätten getestet werden können. Deshalb war die Zeit zur Umsetzung dieses Pakets geringer als ursprünglich geplant.

Beim Umsetzen des Bildverarbeitungsprozesses wurde klar, dass es sich nicht um die optimale Anwendung für CCM mit der heutigen Hardware handelt. Diese Anwendung wäre für Computer mit Multiprozessor-Hardware geeigneter. Diese Art von Hardware steht an der HTA Bern nur bedingt zur Verfügung (nur Dual Prozessor). Deshalb wurde mit dem Fachdozenten Peter Schwab vereinbart, dass eine zweite Anwendung entwickelt wird, die sich einfacher übers Netzwerk verteilen lässt. Als Anwendung wurde das Mandelbrot umgesetzt.

Während der Entwicklung habe ich festgestellt, dass die vorhandene Klassen-Struktur zum Verteilen von Applikationen nicht genügend war. Die Klassen mussten überarbeitet werden, was einen erheblichen Mehraufwand von 8 Tagen bedeutete. Die neue Struktur ist konzeptionell an das RMI von Java angelehnt und wird zur Kommunikation der Zellen eingesetzt.

27 Erkannte (ungelöste) Probleme

Bei der Umsetzung des Cell Computing Model konnte ich auf grosse Erfahrung im Bereich der Entwicklung von Frameworks mit C++ zurückgreifen. Das hat dazu beigetragen, dass schon zu Beginn wenige konzeptionelle Probleme aufgetreten sind und ein umfangreicher Prototypen entwickelt werden konnte. Kleinere Probleme sind trotzdem aufgetreten:

27.1 Technische Probleme

Wie bereits angesprochen, war die Entwicklung der Bewegungserkennung nicht befriedigend, da es vor allem an Hardware zum Testen und Durchführen von Messungen fehlte. Trotzdem konnten gewisse Erkenntnisse bei der Umsetzung gewonnen werden. Somit war diese Beispielimplementation der Bewegungserkennung nicht umsonst.

Zum Umsetzen und Testen der anderen Applikationen gab es keine namhaften Zwischenfälle. Die Hardware und Software-Umgebung an der HTA Bern war gut zu gebrauchen.

28 Erreichte Ziele

Das Ziel ein Framework zu entwickeln, mit welchem die Entwicklung von Cell Computing Model Applikationen vereinfacht wird, wurde erreicht.

Das Produkt befindet sich nun in der Projektphase [Test], und kann erstmals für Applikationen eingesetzt werden. Durch eine ausführliche Beschreibung des Frameworks und die erarbeiteten Beispielimplementationen wurde der Grundstein gelegt, damit andere Personen am Projekt mitarbeiten können.

Die Spezifikation wurde ebenfalls erweitert und verfeinert. Durch den Prototypen gewonnene Erkenntnisse haben zu einer Verbesserung und genaueren Beschreibung des Cell Computing Models beigetragen.

Folgende Ziele wurden erreicht, bzw. nicht erreicht:

Ziel-Nr.	Beschreibung	Erreicht
1	Erarbeiten einer Spezifikation zum Cell Computing Model.	Vollständig
2	Sicherheitskonzept. Erkennen der Zellen mit *finger print* Technik.	Vollständig
3	Beispiel-Implementation eines Bildverarbeitungs-prozesses mit dem erstellten Framework.	Vollständig
4	Erweiterung des Systems mit mobilen Zellen, welche sich autonom im Netz verteilen können (fakultativ).	nur theoretisch.
5	Gebrauch von Petrinetz-Strukturen zur Beschreibung von Prozessen (Theoretisch erarbeiten)	Vollständig.

Tabelle 17: Erreichte Ziele der Diplomarbeit 05/06

28.1 Das Zeus-Framework

Die genaue Beschreibung und Erklärung zum Framework ist im entsprechenden Kapitel zu finden. Hier soll nur kurz erläutert werden, welcher Produkteigenschaften erreicht wurden.

Durch den Einsatz des Zeus-Frameworks ergeben sich folgende positive

Auswirkungen auf die Entwicklung von Software-Produkten:

- Hohe Wiederverwendbarkeit der Komponenten. Das ermöglicht hervorragendes Software-Recycling.
- Dezentrale Entwicklung von Softwarekomponenten. Durch klare Schnittstellen können komplexe Systeme dezentral entwickelt werden.
- Stufenweises Entwickeln. Das System muss nicht schon beim Projektstart ins Detail geplant werden. Komponenten können auch später spezifiziert, entwickelt und integriert werden.
- Erweiterbarkeit der Software. Durch sich ändernde Kundenwünsche kann die Software auch später noch einfach erweitert werden.
- Durch die flexible Konfiguration kann die Software nach Kundenwunsch angepasst werden. Es können beliebige Komponenten in den Objektbaum eingehängt oder entfernt werden.
- Gute Wartbarkeit von komplexen Systemen. Einzelne Komponenten können ausgewechselt oder neu geschrieben werden, ohne dass das gesamte System neu bearbeitet werden muss.

Das Zeus-Framework kann auf sehr verschiedene Arten eingesetzt werden:

- Durch seine modulare Struktur eignet es sich gut für grosse Projekte. Mehrere Entwickler können gleichzeitig an einem Produkt arbeiten. Integration von Softwarekomponenten ist sehr einfach möglich.
- Einsatz für hochdynamische, verteilte Anwendungen, bei denen künstliche Intelligenz zum Einsatz kommt.
  - Neuronale Netze mit Zellen.
  - Evolutionäre Algorithmen
  - Fuzzy Logik
- Es eignet sich für parallele Verarbeitung von Daten (Clustering).
- Es kann zur Entwicklung von Anwendungen dienen, bei denen Java oder C# aus Grund der schlechteren Effizienz nicht eingesetzt werden kann (Bsp. bei hardwarenaher Software)
- Bedien- und Kommunikationssoftware für Maschinensteuerungen

28.1.1 Einfachheit

Die Konfiguration ist einfach und transparent, da sie der statischen Softwarestruktur entspricht. Jede Komponente im System findet sich auch in der Konfiguration wieder. Die Programmierung mit den Klassen des Zeus-Frameworks braucht C++ oder Java Kenntnisse. Zwar braucht es für einzelne Konzepte Zeit zur Einarbeitung, es wird aber kein Ingenieur-Wissen verlangt[7].

Bei der Programmierung von Zellen müssen nur die wesentlichen Schritten gemacht werden. Die Programmierung könnte noch durch Komponenten verbessert werden, welche die Kommunikation der Zellen besser steuern (zum Beispiel Petrinetze).

28.1.2 Schnelligkeit

Zum Entwickeln einer CCM-Anwendung brauchen nur die Zellen entwickelt zu werden. Diese sind in Bibliotheken (Shared Libraries) zu implementieren. Durch das Verwenden von Hilfsklassen ist die Implementation relativ einfach und schnell. Auch die Konfiguration in XML ist durch die übersichtliche Strukturierung schnell gemacht.

28.1.3 Performance

Die Performance des Zeus-Frameworks ist gut. Die Klassen sind zwar vor allem auf die Sicherheit optimiert, aber dank dem direkten Umsetzen in Maschinencode, sehr schnell (C++). Der Code im Framework enthält noch sehr viele Debug-Ausgaben. Durch das Entfernen von unwichtigen Ausgaben kann die Performance noch verbessert werden.

Das RMI (Remote Method Invocation) in C++ hat auch noch Potential zur Verbesserung. Der Objekt-Header der Daten-Serialisierung könnte noch verkleinert und optimiert werden. Dadurch müssten weniger Daten übers Netzwerk übertragen werden.

7 Ingenieur-Wissen braucht es dann, wenn künstliche Intelligenz in die Zellen eingebaut werden sollen.

A Applikationen des Framework Zeus

Das Zeus-Framework umfasst diverse Bibliotheken, Code-Module und Programme. In diesem Kapitel werden die Systemkomponenten des Frameworks erklärt. Weiter wird beschrieben, wie das Framework konfiguriert werden kann.

A.1 Bibliotheken

Die Bibliotheken werden zum Entwickeln und Linken von Applikationen gebraucht. Alle Klassen sind in die entsprechenden Bibliotheken enthalten:

API	
`./lib/zeusbase.lib`	Diese Bibliothek beinhaltet alle Basisklassen zum Entwickeln mit dem Zeus-Framework: • Systemklassen (Netzwerk, Threading, Sockets, Datentypen, Streaming, Serialisierung) • Remote Method Invocation (RMI) • Module Object Model (MOM) • Cell Computing Model • Klassen für künstliche Intelligenz • und viele mehr
`./lib/zeusdev.lib`	Diese Bibliothek dient der Entwicklung von Code-Modulen. Sie beinhaltet Hilfsklassen für Modul-Sessionen und Zellenarchitektur.
`./lib/cppunit.lib`	Zusammengestellte Bibliothek zum Testen der Applikationen. Der Source stammt aus dem Open Source Projekt CPPUNIT (http://sourceforge.net/projects/cppunit).

A.2 Code-Module

Es gibt bereits entwickelte Code-Module die standardmässig zum Framework gehören.

API	
`./bin/modules/XML_Service.so`	Dieses Code-Modul beinhaltet die Anbindung an den XML Parser von Apache (Xercesc) [XML02]
`./bin/modules/LOG_Service.so`	Dieses Code-Modul beinhaltet die Implementation und Anbindung des Logging-Dienst log4cxx von Apache [LOG4CXX]
`./bin/modules/NameServer.so`	Namensdienst für Remote Method Invocation und Remote Objects.

A.3 Programme und Skripts

Folgende Applikationen und Skript-Dateien werden mit dem Framework ausgeliefert

API	
`./bin/zeus`	Diese Datei ist eine Implementation des Zeus-Framework für die Konsole. Mit ihr können logische Einheiten aufgebaut werden. • Clustering • Verteilte Applikationen • Server Dienste
`./bin/mandelbrot`	Eine Beispielimplementation zur Darstellung von Mandelbrot-Mengen. Die Applikation benötigt das Code-Modul `./bin/modules/MandelbrotCellModule.so`, welches die Mandelbrot-Menge berechnet.
`./bin/modec`	Eine Beispielimplementation zur Berechnung und Darstellung von Bewegungen in einem Film. Die Applikation benötigt das Code-Modul `./bin/modules/MotionDetectModule.so`, welches

API	
	den Film lädt und verarbeitet.
`./bin/zeustest`	Testapplikation zum Testen der Basisklassen.
`./run_nameserver`	Startet den Namensdienst, welcher zum Registrieren von Remote Objects (Kommunikationskanäle der Zellen) dient.

A.4 Benutzerhandbuch zum Framework [zeus]

Dieses Kapitel dient als Referenz zur Konfiguration und den Gebrauch der Applikation `./bin/zeus`. Diese Applikation ist reine Konsolen-Applikation und kann sehr einfach zum Erstellen von Clustering und Server-Diensten verwendet werden.

A.4.1 Konfiguration

Die Applikation `./bin/zeus` wird durch 2 Dateien für den benötigten Dienst konfiguriert. Die erste Datei ist die Konfigurationsdatei des Frameworks. Sie beinhaltet Angaben und Einstellungen zum Framework und kann durch kundenspezifische Einstellungen erweitert werden. Standardmässig heisst die Datei `zeus.properties`.

API	
`zeus.LibraryManager`	Dient als Angabe, welche Systemkomponenten (Services) direkt vom Framework geladen und verwaltet werden sollen. Diese Systemkomponenten werden hier als eindeutige Namen angegeben. Die Angabe ist eine Aufzählung (kommagetrennt).
`zeus.LibraryManager.<Name>`	Die einzelnen Systemkomponenten werden dann mit dieser Einstellung konfiguriert. Dabei wird der Namen der Bibliothek (Code Modul) in der Einstellung angegeben, welche vom Framework geladen werden soll (Platzhalter `<Name1>`). Rechts muss dann ein Pfad zugewiesen

API

	werden, welcher dem Bibliothekspfad entspricht.
`zeus.LibraryManager.Security`	Ist diese Einstellung =1, werden alle Code-Module mit einem Sicherheitsdienst verifiziert. Nur registrierte Module werden zugelassen.
`zeus.LoggerManager`	Diese Angabe setzt den gewünschten Logging-Dienst.
`zeus.LoggerManager.ConfigFile`	Der Logging-Dienst braucht auch Einstellungen. Hier wird angegeben, welche Einstellungen verwendet werden sollen. Es kann ein Dateinamen angegeben werden, oder mit der Angabe `BasicConfiguration` wird eine Standardkonfiguration geladen.

Ein Beispiel soll die Anwendung der Konfiguration verdeutlichen:

Beispiel von zeus.properties

```
#-------------------------------------------------------------
# ZEUS CONFIGURATION FILE
#-------------------------------------------------------------
# This file is used to configure the zeus framework
#-------------------------------------------------------------


#-------------------------------------------------------------
# List of all system services. The services are
# comma separated
zeus.LibraryManager=XML_Service,LOG_Service
zeus.LibraryManager.Security=0

# Linux config
zeus.LibraryManager.XML_Service=./modules/XML_Service.so
zeus.LibraryManager.LOG_Service=./modules/LOG_Service.so

zeus.LoggerManager=LOG_Service
zeus.LoggerManager.ConfigFile=zeus.log.properties
```

Listing 6: Beispiel einer Framework Konfiguration

Die zweite Datei dient dem Erstellen des Module Object Model (MOM). Diese Datei ist ein XML Dokument und heisst standardmässig `zeus.xml`.

Der Aufbau ist je nach Anwendung sehr verschieden. Zum Beispiel die

Testanwendung zum Zeus-Framework:

Beispiel Testapplikation

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<zeus ClassName="TSystemManager">
  <LocalPipeRegistry ClassName="TLocalPipeRegistry"
    RegistryPort="7278"
    RegistryAddress="127.0.0.1"/>
  <Services ClassName="TModule" CreateChildren="0">
    <XML_Service ClassName="TModule"/>
    <LOG_Service ClassName="TModule"/>
  </Services>
  <CellSystem1 ClassName="TModule" CreateChildren="1">
    <WorkCell ClassName="TCellEnvironment"
       CreateChildren="1"
       ModuleLibrary="WorkCellModule"
       Security="1">
      <InputPipe Name="Pipe1"/>
      <InputPipe Name="Pipe2"/>
      <Cell ClassName="TCell" CreateChildren="1">
        <Kernel ClassName="TWorkCell" CodeModule="WorkCellModule"/>
      </Cell>
      <Survey ClassName="TCellSurvey"/>
    </WorkCell>
  </CellSystem1>
</zeus>
```

Listing 7: Beispiel einer MOM Konfiguration. Die Testapplikation erzeugt eine Arbeitszelle, die WorkCell.

A.4.2 Kommandozeilen Parameter

Die Zeus-Framework Applikation kann durch Kommandozeilen-Parameter zusätzlich beeinflusst werden. Folgende Parameter sind zulässig:

API	
`-conf=<file>`	Durch diese Angabe wird nicht die MOM-Konfiguration `zeus.xml` gelesen, sondern die Datei `<file>`.
`-prop=<file>`	Durch diese Angabe wird nicht die Framework-Konfiguration `zeus.properties` gelesen, sondern die Datei `<file>`.

API	
`-reg`	Registriert ein Code-Modul bei dem Sicherheitsdienst. Dieses Kommando braucht noch die Angaben `-file` und `-module`.
`-unreg`	Entfernt ein Code-Module von der Datenbank des Sicherheitsdiensts. Dieses Kommando braucht die Angabe `-module`.
`-file`	Mit dieser Angabe kann ein Code-Modul mit Pfad angegeben werden.
`-module`	Gibt den Namen an eines Code-Moduls. Dieser ist muss dem Dateinamen des Code-Moduls entsprechen (ohne Endung)
`-list`	Zeigt alle registrierten Code-Module an.
`--version`	Zeigt die Version der Applikation an
`--help` `-?`	Zeigt die Beschreibung der Kommandos in der Konsole an.

A.5 Der Namensdienst

Um eine Applikation (mit oder ohne Zellen) auf verschiedene Computer zu verteilen, braucht es einen Dienst, bei welchem sich die Kommunikationsobjekte der Teilapplikationen registrieren können. Ein solcher Dienst ist der Namensdienst.

Der Namensdienst dient der Registrierung von Objekten, die auf entfernten Computersystemen leben. Dieser Dienst ist eine sehr wichtige Komponente, wenn mit verteilten Applikationen gearbeitet werden möchte. Durch das Registrieren sind diese Objekte für andere Anwendungen zugänglich und können mit dem RMI für C++ über Netzwerk verwendet werden.

Der Namensdienst ist im Code-Module `./bin/modules/NameServer.so` implementiert. Die Konfiguration des Dienst, muss in der Datei

`./nameserver.xml` vorgenommen werden. Hier muss die IP-Adresse und der TCP-Port eingetragen werden.

Durch die Skript-Datei `./nameserver` kann der Dienst gestartet werden.

Einstellen des Namensdiensts

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<CellReg ClassName="TSystemManager">
  <NameServer ClassName="TNameServer" CodeModule="NameServer"
     Address="127.0.0.1"
     Port="7278"/>
</CellReg>
```

Listing 8: Einstellen des Namensdienst. In dieser XML Datei werden Adresse und Port eingestellt.

A.5.1 Konfiguration der Zellen

Damit die Zellensysteme diesen Dienst verwenden können, müssen Adresse und Port auch bei den entsprechenden Zeus-Applikationen eingestellt werden. Dies geschieht durch den Eintrag bei der lokalen Registrierungsstelle.

Einstellen der Zellen

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<zeus ClassName="TSystemManager">
  <LocalPipeRegistry ClassName="TLocalPipeRegistry"
    RegistryPort="7278"
    RegistryAddress="127.0.0.1"/>
...
</zeus>
```

Listing 9: Einstellung bei einer Zeus-Applikation. In dieser XML Datei werden Adresse und Port des Namensdienst angegeben.

B Messungen vom Klonverhalten

B.1 Klonen von Zellen

Das Zeus-Framework verwendet den stabilen Klon-Algorithmus, welcher auch bei instabilen Arbeitslast der Zellen ein Optimum zwischen Anzahl Zellen und Auftragseingang (Work load) findet. Siehe [CCM1].

Das Klonen ist der elementarer Bestandteil zur Wachstumskontrolle der Zeus-Zellen. Die Kontrolle wird durch eine Zellenüberwachung in der Zellenumgebung betätigt. Der Klon-Algorithmus misst die gesamte Auslastung einer Zellenumgebung.

Der Algorithmus benötigt folgende Eingabeparameter:

- Anzahl Meldungen eines Zellensystems zu einem beliebigen Zeitpunkt t: $d(t)$
- Maximale Grenze des Meldungseingangs (Grösse der Meldungsqueue) N_{MAX}
- Reaktionsgrenze N_{react}
- Obere Schranke der Messzeit T_{MAX}
- Untere Schranke der Messzeit T_{MIN}

B.2 Abbauen von Zellen

Das zweite Verfahren steuert den Abbau von Zellen. Dabei wird jeder neu erstellten Zelle die Lebenskraft 100 zugewiesen (bedeutet 100%).

Der Algorithmus verwendet folgende Eingabeparameter:

- Lebenskraft der Zelle in Abhängigkeit der Zeit t: $g(t)$
- Erhöhung der Lebenskraft L_{add}
- Verringerung der Lebenskraft L_{sub}
- Faktor für Wahrscheinlichkeit des Abbaus ω

B.3 Messungen

In diversen Versuchen wurden empirisch folgende Werte für Zellen mit variabler Eingangslast ermittelt[8]:

Parameter	*Wert*
T_{MAX}	1.0 [sec]
T_{MIN}	0.1 [sec]
N_{MAX}	20 [jobs]
N_{react}	10 [jobs]
L_{add}	1.4
L_{sub}	1
ω	7

8 Die Messung erfolgte auf einem Laptop Sony PCG-FX290K mit 1GHz Prozessor und 512MB RAM. Als Systemsoftware wurde SUSE Linux 9.1 mit Kernel 2.6.4 verwendet (GCC 3.3.3)

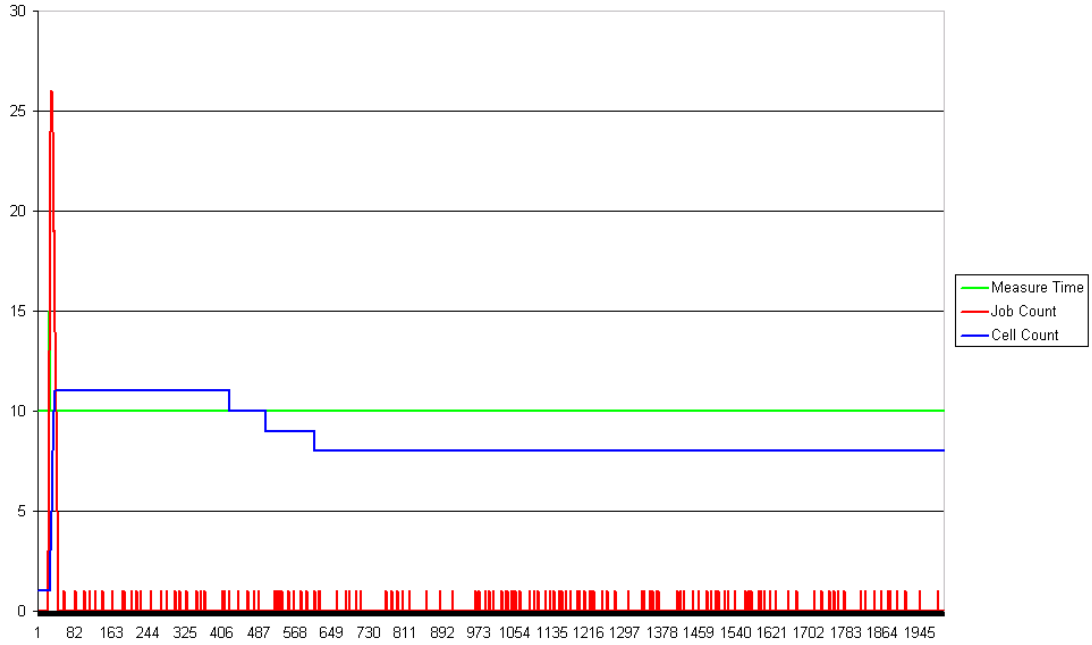


Abbildung 42: Messung eines stabilen Systems, mit variabler Eingangslast. Measure Time zeigt die Messdauer (Dargestellt mit Faktor 100), Job Count ist die Grösse des Meldungseingangs und Cell Count zeigt auf, wie viele Zellen aktiv sind.

Das System wird mit folgenden Parameter instabil:

Parameter	***Wert***
T_{MAX}	1.0 [sec]
T_{MIN}	0.1 [sec]
N_{MAX}	20 [jobs]
N_{react}	15 [jobs]
L_{add}	1
L_{sub}	1.2
ω	7

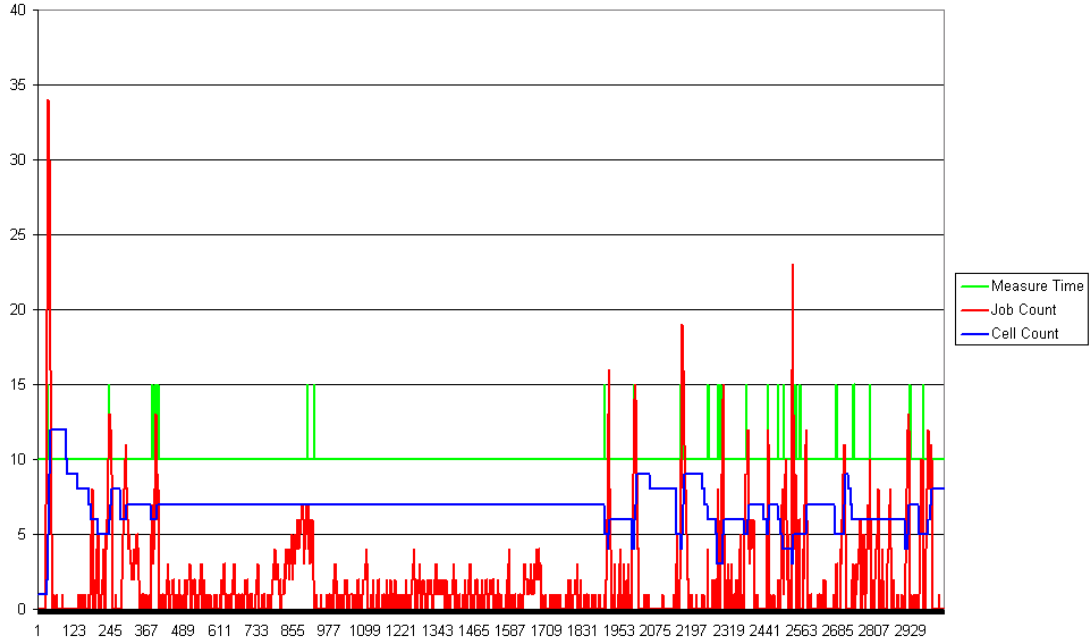
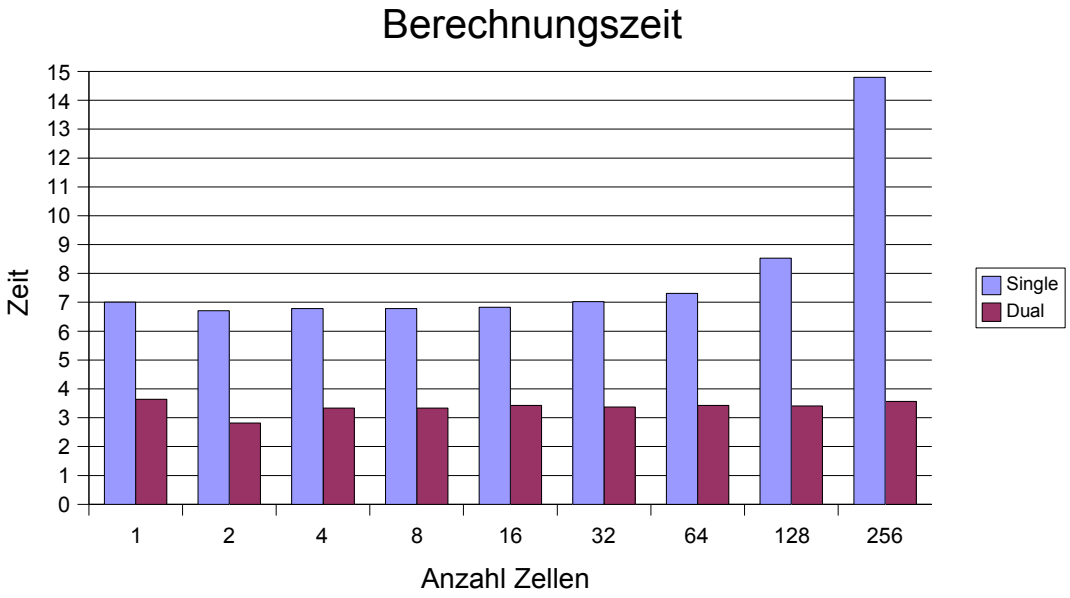


Abbildung 43: Messung eines instabilen Systems. Klar ersichtlich ist der Unterschied zur vorangehenden Abbildung. Weil sehr viele Zellen erstellt und wieder abgebaut werden, ist die Performance beim Verarbeiten der Arbeitspakete schlechter (Measure Time Kurve)

C Messungen mit Shared Memory

Zum Messen einer verteilten Applikation im Sinne von mehreren Zellen, die auf einem Computer laufen, wurde die Applikation `modec` verwendet. Diese Applikation detektiert in einem Film Bewegungen.

Die `modec` Anwendung mit dem Zeus-Framework ist für Multiprozessor-Systeme geeignet, welche auf einen Shared Memory zugreifen können. Der Film muss nicht dupliziert werden, sondern die Zellen arbeiten direkt am gleichen Bilder-Stream.



Auffallend ist, dass der Dual-Prozessor mit 2 Zellen ca. 22% schneller ist als nur mit einer einzigen Zelle. Der Single-Prozessor ist auch etwas schneller, mit 4% fällt dies aber sehr moderat aus. Interessant ist auch, dass beim Single-Prozessor die Belastung ab 64 Zellen extrem zu nimmt, während der Dual-Prozessor stabil bleibt.

C.1 Konfiguration

Hardware

Single Processor	htibe2ci12-006	2.533GHz	1024MB RAM	Ubuntu
Dual Processor	pinguin	3.601GHz	2048MB RAM	Ubuntu
Anzeige PC	htibe2ci12-005	2.533GHz	1024MB RAM	Ubuntu

Einstellungen des Modec

Blockgrösse	16 Pixel
Blocks pro Fragment	200 Blocks
Max. Bewegung	10 Pixel
Zellenüberwachung Max	2 Sekunden
Zellenüberwachung Min	1 Sekunde

Software

Zeus Version	0.2.5
QT Version	3.xx
Verbindung	ssh -X (für X-Server)

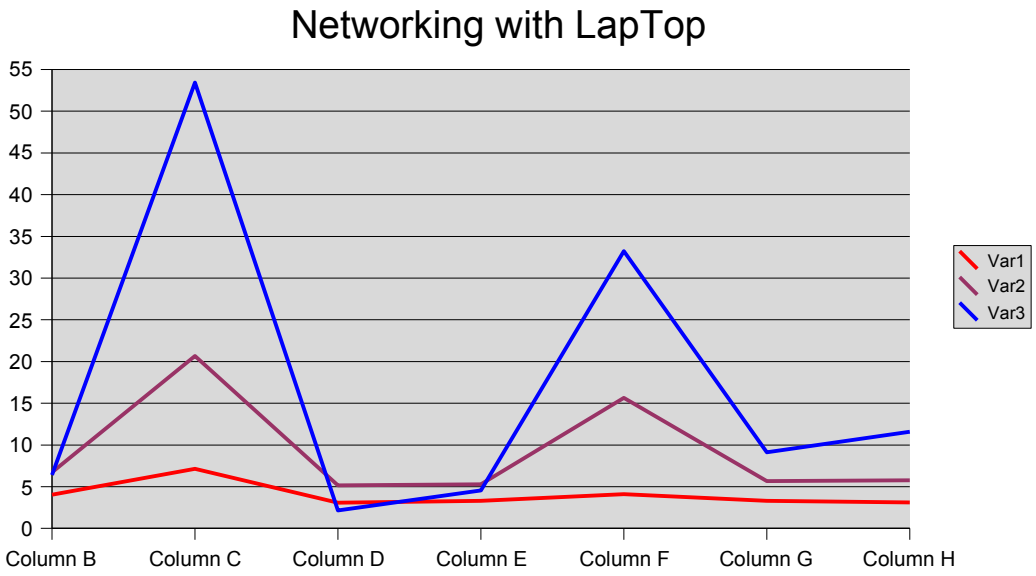
D Messungen mit verteilten Applikationen

Mit der Mandelbrot Menge wurden Messungen durchgeführt, die das verteilen einer Applikation auf verschiedene Computer untersucht. Dabei wurden verschiedene Konfigurationen untersucht:

Variante	*Beschreibung*
Var1	7 Arbeits PC's a 2.533GHz und ein Laptop 1GHz zum Anzeigen der Daten
Var2	1 Arbeits PC a 2.533GHz und ein Laptop 1GHz zum Anzeigen der Daten
Var3	Inprozess auf dem Laptop 1GHz.
Var4	Inprozess auf einem PC 2.533GHz
Var5	6 Arbeits PC's a 2.533GHz und ein PC 2.533GHz zum Anzeigen der Daten
Var6	Inprozess kombiniert mit 6 Arbeits PC's a 2.533GHz

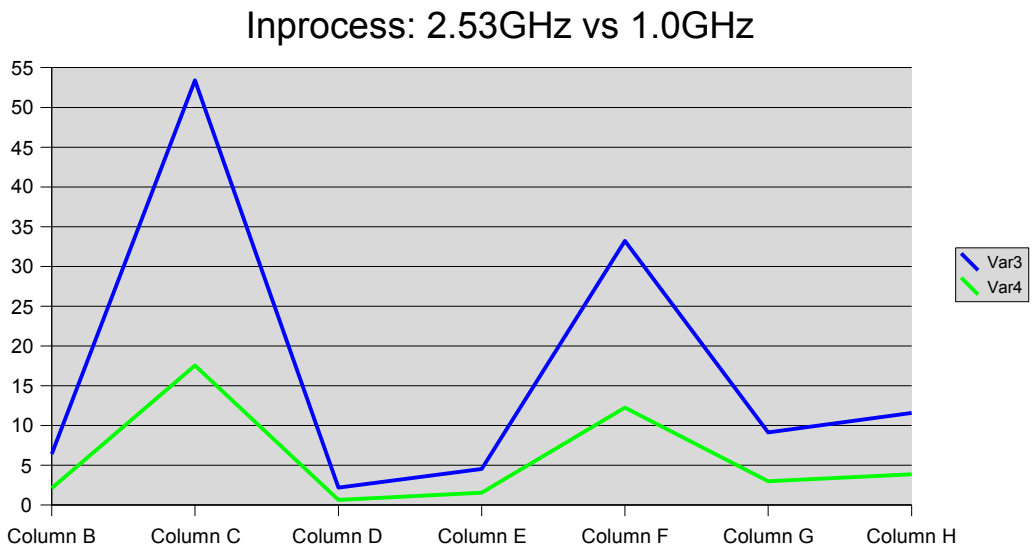
D.1 Unterschied der Performance (Inprozess oder Verteilen)

Der Unterschied von Variante 1 und Variante 3 zeigt den Aufwand, wenn entweder alles auf einem Rechner gerechnet werden muss (3) oder wenn die Arbeit auf mehrere PC's verteilt werden kann. Bereits bei einem zusätzlichen PC (Variante 2) ist eine wesentliche Steigerung zu spüren.

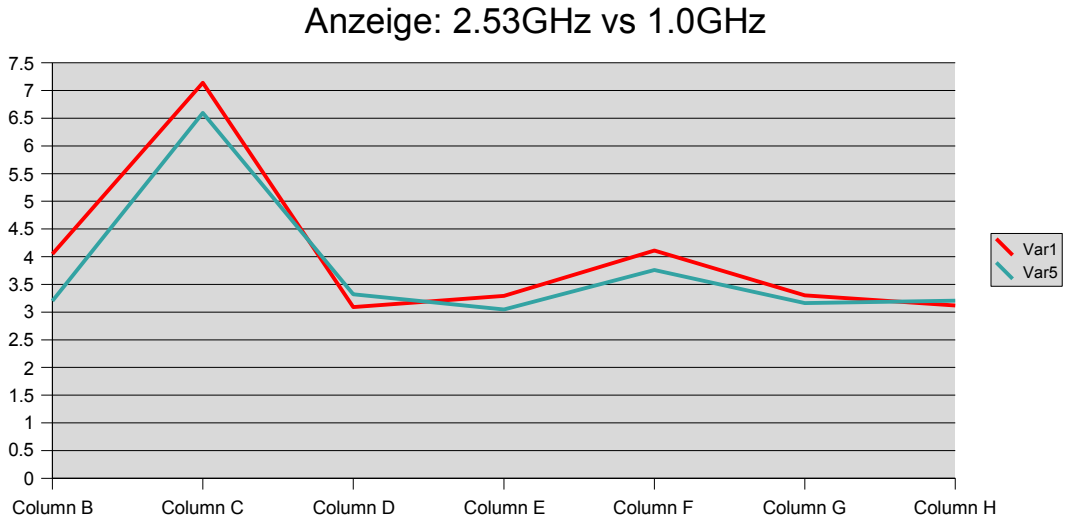


D.2 Vergleichsmessung

Dies ist eine Vergleichsmessung zwischen einem Laptop mit 1GHz Prozessor (Variante 3) und einem PC mit 2.533GHz Prozessor (Variante 4). Zudem ist der Arbeitsspeicher beim PC doppelt so gross. Man sieht deutlich, dass der PC ca. 3 x schneller ist als der Laptop. Die Inprozess-Varianten berechnen die gesamte Arbeit selber und zeichnen auch das Resultat



Die nächste Messung zeigt den Verbrauch beim Darstellen der Bilder auf. Dabei ist deutlich zu erkennen, das der Unterschied zwischen dem schnelleren PC zum langsamen PC nicht gross ist.



D.3 Konfiguration

Hardware und Software Konfigurationen sind hier enthalten.

Hardware

Laptop	Sony Vaio	1GHz	512MB RAM	SUSE 9.1
Htibe2ci12-005	Pentuim 4	2.533GHz	1024MB RAM	Ubuntu
Htibe2ci12-006	Pentuim 4	2.533GHz	1024MB RAM	Ubuntu
Htibe2c5-006	Pentuim 4	2.533GHz	1024MB RAM	Ubuntu
Htibe2c5-007	Pentuim 4	2.533GHz	1024MB RAM	Ubuntu
Htibe2c5-010	Pentuim 4	2.533GHz	1024MB RAM	Ubuntu
Htibe2c5-011	Pentuim 4	2.533GHz	1024MB RAM	Ubuntu
Htibe2c5-012	Pentuim 4	2.533GHz	1024MB RAM	Ubuntu

Einstellungen der Mandelbrot Applikation

Max Iterations	3500
Max Range	1000
Zoom	3000 %

Weg

X: 0.73580217851
Y: 0.19671588244

Software

Zeus Version	0.2.5
QT Version	3.xx

Folgende Sammlung der Bilder zeigt den Verlauf, welcher beim Messen der Performance massgebend war. Die Positionen in der Mandelbrot Menge sind aus der Tabelle zu entnehmen.

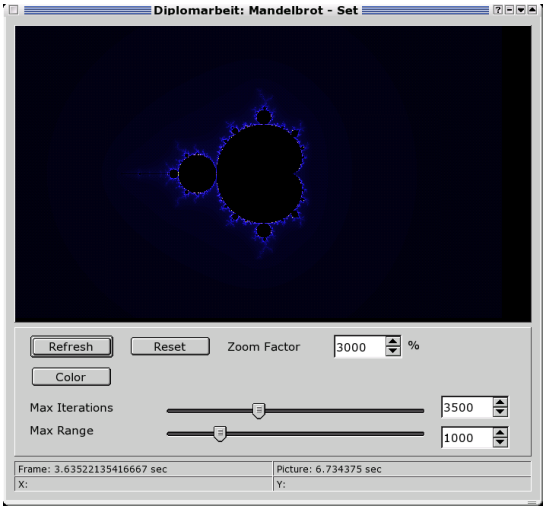


Abbildung 47: Step 1 (Mandelbrot)

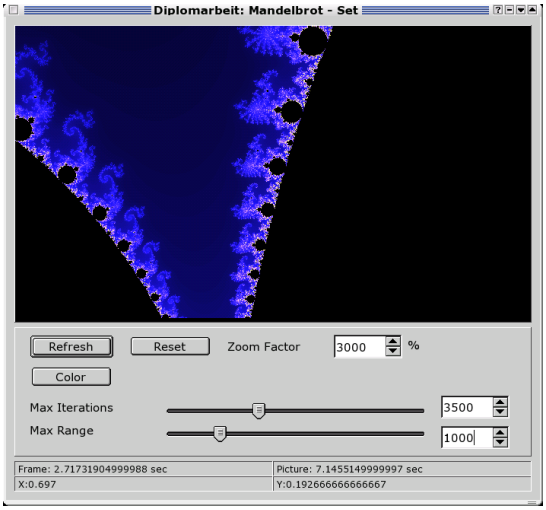


Abbildung 48: Step 2 (seehorse valley)

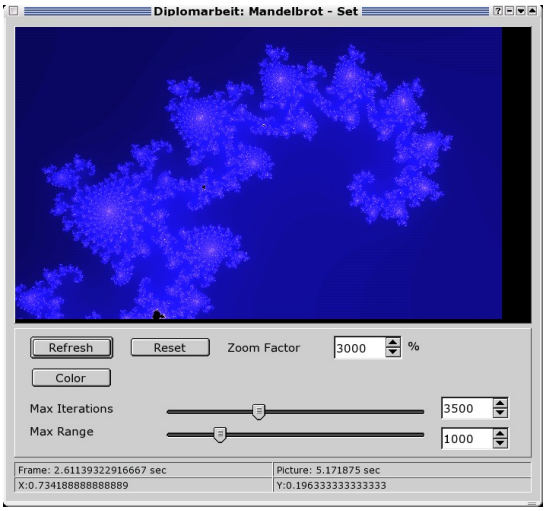


Abbildung 49: Step 3 (See horse tail)

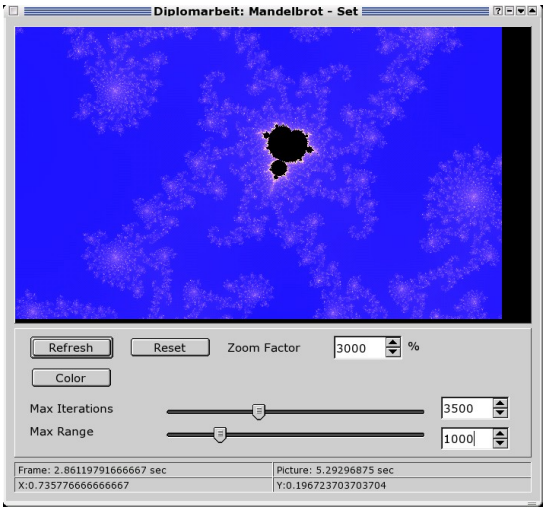


Abbildung 50: Step 4 (double hook with satelite)

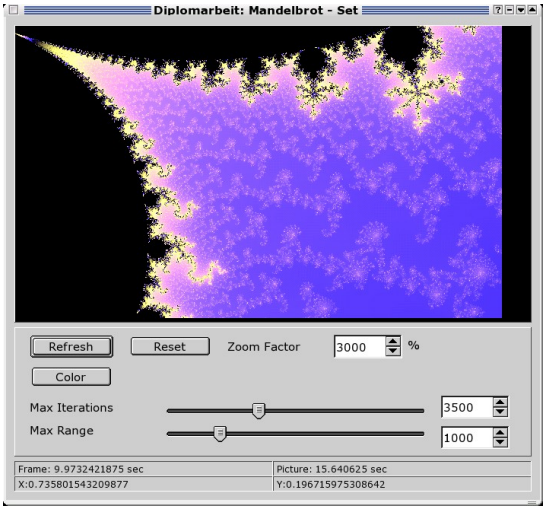


Abbildung 51: Step 5 (Seehorse valley of satelite)

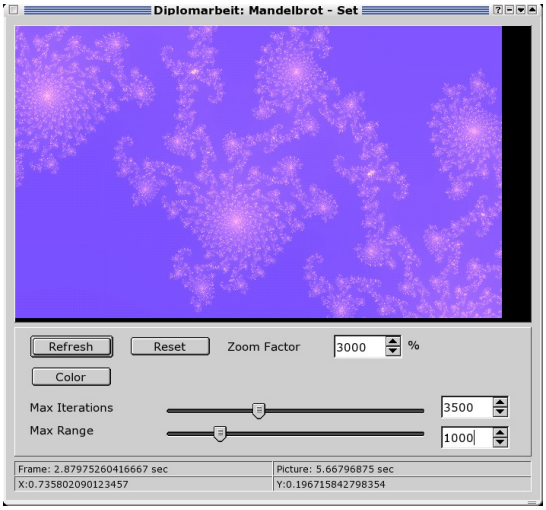


Abbildung 52: Step 6 (seehorse of satelite)

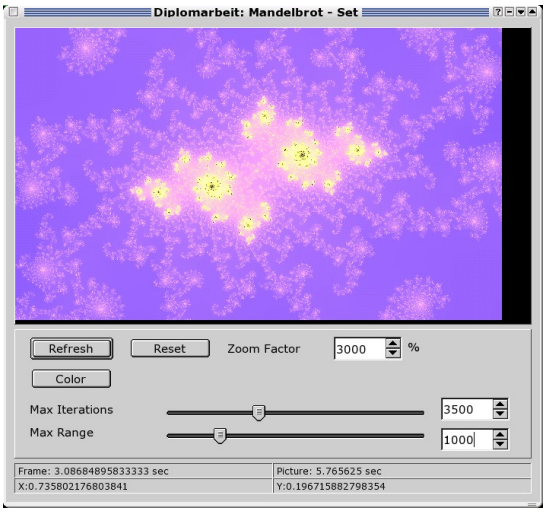


Abbildung 53: Step 7 (Julia Islands)

E Versionsgeschichte

Änderungen am Dokument sind hier festgehalten

Version	*Beschreibung*	*Datum*
1.0	Freigabe für Diplomarbeit	23.12.2004

Literaturverzeichnis

[GEOK2] Diverse GEO kompakt. Das Wunder Mensch 2005

[MSENC] Microsoft Microsoft Encarta 2001

[CCM1] Hadorn, Benjamin Cell Computing Model 2005

[XML01] W3C Extensible Markup Language (XML)

[STWIN] Benjamin Hadorn Programmieranleitung zum StuderWINFrame 2003

[XML02] Apache

[CCM1] Hadorn, Benjamin Cell Computing Model 2003

[PTHREAD] Xavier Leroy Manual Linux Threads

[POSIXTH] Ulrich Drepper, Ingo Molar The Native POSIX Thread Library for Linux, 2005 2005

[DIP-ISWP1] P. Schwab Diplomarbeit 2005 zum Thema Cell Computing Model 2005

[QTDOC] Trolltech.com

[LOG4CXX] Apache Logging Services Project @ Apache 2005